北京大学

PEKING UNIVERSITY

操作系统

Operating Systems

课 程 笔 记

Course Notes

学院 : 北京大学计算机学院

学期 : 2026 年春季学期

主讲 : 陈向群 教授

2026 年 6 月 12 日

北京大学操作系统课程笔记

本课程笔记按学期讲义顺序整理

覆盖讲义思考题、典型流程与 XV6 代码阅读题

目　录

第 0 讲：课程简介 2

0.1 课程定位与学习方法 2

0.2 如何学好操作系统 2

0.3 HelloWorld 背后的操作系统机制 3

0.4 课程实验与源码阅读 5

0.5 启动流程与课程主线 5

0.5.1 从固件到内核：启动链路的分层理解 5

0.5.2 多核启动中的栈安排 6

0.5.3 `start` 运行在什么模式里？ 7

0.5.4 从 `main` 看内核初始化 8

0.6 实验体系与工程能力 9

0.6.1 为什么要做操作系统实验 9

0.6.2 XV6 为什么适合作为学习材料 9

0.6.3 做实验时应该记录什么 11

0.7 本讲总结 11

0.7.1 需要带走的主线 11

0.7.2 复习检查表 11

第 1 讲：操作系统概述 13

1.1 本讲导读 13

1.2 开篇问题 13

1.3 操作系统的定义与作用 14

1.3.1 一句话与三层含义 15

1.3.2 三种作用：虚拟机 / API / 资源管理器 15

1.3.3 设计与实现目标 15

1.4 三种认知视角 16

1.4.1 资源管理视角（自底向上看） 16

1.4.2 进程视角（动态看） 18

1.4.3 虚机器视角（结构看） 18
1.5 操作系统的特征 19
1.5.1 并发（Concurrency） 19
1.5.2 共享（Sharing） 20
1.5.3 虚拟（Virtual） 20
1.5.4 随机性（不确定性） 21
1.6 发展历程及启示 21
1.6.1 第 1 阶段（1948–1970）：硬件昂贵，人工便宜 21
1.6.2 第 2 阶段（1970–1985）：硬件便宜，人工昂贵 21
1.6.3 第 3 阶段（1981 起）：个人计算时代 22
1.6.4 第 4 阶段（1981 起）：分布式 22
1.6.5 第 5 阶段（1995 起）：移动计算 23
1.6.6 第 6 阶段（2006 起）：云计算 23
1.6.7 第 7 阶段（2000 年代起）：泛在计算 / 物联网 24
1.7 操作系统分类 24
1.7.1 批处理操作系统 25
1.7.2 分时操作系统 25
1.7.3 实时操作系统 26
1.7.4 个人计算机操作系统 26
1.7.5 网络操作系统 27
1.7.6 分布式操作系统 27
1.7.7 嵌入式操作系统 28
1.8 重点小结 28
第 2 讲：操作系统运行环境和运行机制 31
2.1 本讲导读 31
2.2 开篇问题与核心结论 31
2.2.1 八个开篇问题精炼解答 31
2.2.2 与系统调用相关的开篇问题（第 3 页） 34
2.3 本讲大纲 35
2.4 中央处理器与寄存器 36
2.4.1 寄存器分类 36
2.4.2 PSW 的作用 37
2.4.3 x86 控制寄存器 CR0–CR4 37

2.5 保护机制：特权级与状态切换 37
2.5.1 特权级的本质与 EFLAGS 37
2.5.2 特权指令与非特权指令 37
2.5.3 用户态与内核态的切换 38
2.6 中断机制与事件分类 39
2.6.1 中断/异常的概念与特点 39
2.6.2 引入原因（历史） 40
2.6.3 中断与异常的来源（细化） 41
2.6.4 事件分类总表 42
2.7 中断/异常机制的工作原理 43
2.7.1 硬件视角：响应过程 43
2.7.2 软硬协同：中断向量表 44
2.7.3 中断响应的 5 步 45
2.7.4 上半部与下半部 45
2.7.5 软件视角：中断处理程序 46
2.7.6 完整流程小结（以打印机中断为例） 46
2.8 IA-32 体系结构对中断的支持 47
2.8.1 基本概念 47
2.8.2 实模式 vs 保护模式 47
2.8.3 IDT 中的四种门描述符 48
2.8.4 段选择符与 IDT/GDT 联动 49
2.8.5 IA-32 的硬件处理过程（8 步） 49
2.9 系统调用机制 50
2.9.1 系统调用是什么 51
2.9.2 系统调用 / 库函数 / API / 内核函数 51
2.9.3 系统调用机制的设计 52
2.9.4 参数传递的三种方式 53
2.9.5 系统调用的高级语言/汇编视角 54
2.9.6 系统调用的执行过程 56
2.10 Linux 系统调用实现（x86 视角） 56
2.10.1 入口与门描述符 57
2.10.2 系统调用号示例 57
2.10.3 `int $0x80` 的硬件动作（详细两步） 59

2.10.4 system_call 的执行流程 ... 60
2.10.5 SAVE_ALL 宏 ... 60
2.10.6 sys_call_table 与系统调用分派 ... 61
2.10.7 系统调用服务例程示例 ... 62
2.10.8 系统调用的参数传递（汇编视角） ... 63
2.10.9 system_call 关键片段 ... 64
2.10.10 ret_from_sys_call ... 65
2.10.11 系统调用小结 ... 66
2.11 中断发生后操作系统底层的 8 步 ... 67
2.12 机制与策略分离原则 ... 67
2.13 XV6 源代码阅读题（来自第 67 页） ... 69
2.14 本讲小结 ... 70
第 3 讲：进程线程模型 ... 73
3.1 本讲导读 ... 73
3.2 三组开篇思考题 ... 73
3.2.1 怎样理解“进程是对 CPU 的抽象”？ ... 73
3.2.2 进程映像与进程实体 ... 74
3.2.3 进程的组成要素 ... 75
3.2.4 创建进程主要做哪些工作？ ... 76
3.2.5 进程生命周期与状态转换 ... 76
3.2.6 状态转换是否会引发其他转换？ ... 76
3.2.7 进程地址空间是虚拟还是物理？ ... 77
3.2.8 OS 如何描述地址空间？ ... 77
3.2.9 为什么需要线程 ... 77
3.2.10 线程怎么实现，为何有多种方式 ... 78
3.2.11 Pthreads 主要函数 ... 78
3.2.12 中断 / 异常机制与进程线程模型的关联 ... 78
3.2.13 机制 / 策略分离在本讲的体现 ... 79
3.2.14 协程是什么、为什么、怎么用 ... 79
3.2.15 纤程 ... 79
3.3 进程模型基础概念 ... 79
3.3.1 顺序程序与顺序环境 ... 79
3.3.2 多道程序设计 ... 80

3.3.3 并发环境与并发程序 80
3.3.4 进程的定义 81
3.3.5 进程相关的若干区分 82
3.4 进程状态及其转换 83
3.4.1 三种基本状态 83
3.4.2 三态之间的转换 84
3.4.3 创建态、终止态、挂起态 84
3.4.4 五状态与七状态模型 85
3.4.5 Linux 进程状态 86
3.4.6 XV6 进程状态 86
3.5 进程控制块 PCB 87
3.5.1 PCB 概念 87
3.5.2 PCB 的主要内容 88
3.5.3 Solaris 与 Linux 的具体 PCB 89
3.5.4 XV6 PCB 90
3.6 进程地址空间 91
3.6.1 概念与示例 91
3.6.2 ASLR 与重复运行的影响 92
3.6.3 进程地址空间的典型布局 92
3.6.4 /proc/<pid>/maps 与 VMA 93
3.6.5 Linux 内核层面的地址空间 95
3.7 进程队列模型 95
3.8 进程控制 96
3.8.1 原语 96
3.8.2 何时创建 / 何时终止 97
3.8.3 进程创建原语 98
3.8.4 进程撤消原语 99
3.8.5 进程阻塞与唤醒 99
3.8.6 UNIX 系统调用：fork / exec / wait / exit 100
3.8.7 UNIX fork() 实现与 COW 优化 101
3.9 线程模型 103
3.9.1 为什么引入线程 103
3.9.2 应用示例：多线程字处理与 Web 服务器 104

3.9.3 开销的考虑 106
3.9.4 线程的基本概念 107
3.9.5 线程实现的三种方式 108
3.10 协程与纤程 112
3.11 关键词与思维网络 113
3.12 重点小结 114
3.13 XV6 源代码阅读 115
第 4 讲：进程线程调度 118
4.1 本讲导读 118
4.2 三组开篇思考题 118
4.3 大纲与三个层次 121
4.3.1 调度的三个层次 121
4.4 短程调度的三个问题：What / When / How 122
4.4.1 CPU 调度的时机 122
4.4.2 调度过程：进程切换 124
4.5 调度算法的设计目标 127
4.5.1 调度算法的衡量指标 128
4.6 调度算法设计的若干要点 128
4.6.1 进程优先级（数） 128
4.6.2 就绪队列组织 129
4.6.3 抢占式 vs 非抢占式 130
4.6.4 I/O 密集型 vs CPU 密集型 130
4.6.5 时间片 131
4.7 批处理系统中的调度算法 132
4.7.1 先来先服务 FCFS 132
4.7.2 短作业优先 SJF / SRTN 133
4.7.3 最高响应比优先 HRRN 135
4.8 交互式系统中的调度算法 136
4.8.1 时间片轮转 RR 136
4.8.2 虚拟轮转法 Virtual RR 138
4.8.3 优先级调度 139
4.8.4 优先级反转 139

4.8.5 多级队列调度 . . . 140
4.8.6 多级反馈队列 MLFQ . . . 141
4.8.7 公平共享 / 保证调度 / 彩票调度 . . . 142
4.8.8 调度算法对比表 . . . 143
4.9 机制与策略分离 . . . 145
4.10 典型操作系统的调度算法 . . . 146
4.11 Windows 线程调度 . . . 146
4.11.1 总体特征 . . . 146
4.11.2 线程优先级 . . . 148
4.11.3 时间配额 . . . 148
4.11.4 调度器数据结构 . . . 149
4.11.5 调度策略：主动切换 / 抢占 / 配额用完 . . . 150
4.11.6 线程优先级提升与时间配额调整 . . . 151
4.11.7 空闲线程 . . . 154
4.12 多处理器调度 . . . 154
4.12.1 多处理器调度的特征 . . . 155
4.12.2 多处理器调度算法的设计 . . . 155
4.12.3 静态 / 动态进程分配 . . . 156
4.12.4 多核处理器的特殊问题 . . . 156
4.13 实时调度 . . . 159
4.13.1 可调度性 . . . 159
4.13.2 速率单调 RM . . . 160
4.13.3 最早截止时间优先 EDF . . . 161
4.14 openEuler 的多核调度 . . . 161
4.14.1 CPU 调度概念 . . . 161
4.14.2 常见调度算法 . . . 162
4.14.3 CFS 调度 . . . 162
4.14.4 多核处理器系统 . . . 162
4.14.5 单队列 vs 多队列调度 . . . 163
4.15 Linux 调度算法的演化 . . . 163
4.15.1 调度器演化历史 . . . 164
4.15.2 Linux 进程分类 . . . 165
4.15.3 Linux 2.4 的调度器 . . . 165

4.15.4 O(1) 调度器 166
4.15.5 SD（楼梯调度） 167
4.15.6 RSDL 167
4.15.7 CFS：完全公平调度器 168
4.16 重点小结 175
4.17 XV6 源代码阅读 176

第 5 讲：内存管理概述 183
5.1 本讲导读 183
5.2 开篇问题与重要概念 183
5.3 重要概念与本讲大纲 185
5.4 演化过程与基础概念 186
5.4.1 历史演化 186
5.4.2 多道程序设计带来的需求 187
5.4.3 地址空间是对内存的抽象 188
5.4.4 进程地址空间内容 188
5.4.5 XV6 的地址空间 189
5.5 地址重定位与地址保护 190
5.5.1 程序执行前的准备过程 190
5.5.2 逻辑地址 vs 物理地址 191
5.5.3 静态地址重定位 191
5.5.4 动态地址重定位与 MMU 192
5.5.5 地址保护 192
5.5.6 内存管理的基本功能 193
5.6 物理内存管理 194
5.6.1 空闲物理内存的数据结构 194
5.6.2 衡量指标 195
5.6.3 内存分配算法 195
5.6.4 分配 / 回收举例 196
5.6.5 伙伴系统 196
5.6.6 SLAB / SLUB / SLOB 197
5.7 基本内存管理方案 200
5.7.1 各方案对比 201
5.7.2 单一连续区 202

5.7.3 固定分区 . . . 202
5.7.4 可变分区 . . . 203
5.7.5 页式内存管理 . . . 204
5.7.6 段式内存管理 . . . 205
5.7.7 段页式内存管理 . . . 207
5.8 内存“扩充”技术 . . . 208
5.8.1 覆盖技术 Overlaying . . . 208
5.8.2 交换技术 Swapping . . . 210
5.9 重点小结 . . . 212
第 6 讲：虚拟存储技术 . . . 215
6.1 本讲导读 . . . 215
6.2 开篇问题 . . . 215
6.3 本讲大纲 . . . 218
6.4 虚拟页式存储管理基础 . . . 218
6.4.1 存储体系 . . . 218
6.4.2 概念辨析 . . . 218
6.4.3 虚拟页式存储管理基本思想 . . . 219
6.4.4 调页机制：三个核心策略 . . . 220
6.4.5 设计与实现要解决的问题 . . . 220
6.5 页表表项设计 . . . 221
6.5.1 i386 PDE / PTE . . . 222
6.6 多级页表 . . . 222
6.6.1 单级页表的规模问题 . . . 222
6.6.2 两级页表设计思想 . . . 222
6.6.3 两级页表的地址映射 . . . 223
6.6.4 Core i7 四级页表 . . . 223
6.7 反转 / 反置页表 . . . 224
6.7.1 实现机制 . . . 225
6.7.2 应用 . . . 225
6.8 操作系统的存储抽象与 MMU . . . 226
6.9 地址转换流程 . . . 226
6.10 快表 TLB . . . 227

6.10.1 引入 . . . 227
6.10.2 TLB 是什么 . . . 228
6.10.3 TLB 的使用与高速缓存的关系 . . . 229
6.11 缺页异常处理 . . . 229
6.12 驻留集管理 . . . 230
6.13 页面置换算法 . . . 231
6.13.1 置换问题 . . . 231
6.13.2 页框锁定 . . . 233
6.13.3 各种置换算法 . . . 233
6.13.4 影响缺页次数的因素 . . . 240
6.14 工作集模型 . . . 241
6.14.1 工作集与驻留集 . . . 243
6.14.2 工作集算法 . . . 243
6.14.3 缺页率算法 . . . 244
6.14.4 算法小结 . . . 246
6.15 清除策略与加载控制 . . . 247
6.15.1 清除策略—分页守护进程 . . . 247
6.15.2 页缓冲技术 . . . 248
6.15.3 加载控制 . . . 249
6.16 内存映射文件 . . . 249
6.16.1 基本思想 . . . 249
6.16.2 mmap 函数原型 . . . 250
6.16.3 flags 详解 . . . 251
6.16.4 虚拟存储管理全貌 . . . 251
6.16.5 共享对象 vs COW 私有对象 . . . 252
6.16.6 fork 与 COW . . . 253
6.16.7 内存映射文件的应用 . . . 254
6.17 Windows 虚拟存储管理 . . . 255
6.17.1 Intel x86 虚拟内存机制 . . . 255
6.17.2 内存管理器 . . . 255
6.17.3 地址空间布局与磁盘后备 . . . 256
6.17.4 地址转换机制 . . . 256
6.17.5 缺页异常处理 . . . 257

6.17.6 页目录自映射机制 . . . 258
6.17.7 工作集 . . . 259
6.17.8 用户空间内存分配方式 . . . 259
6.17.9 物理内存管理 . . . 262
6.18 重点小结 . . . 265
6.19 XV6 源代码阅读 . . . 267

第 7 讲：文件系统 1 . . . 279

7.1 本讲导读 . . . 279
7.2 开篇问题 . . . 279
7.3 文件与文件目录 . . . 281
7.3.1 文件的基本概念 . . . 281
7.3.2 文件类型 . . . 282
7.3.3 文件控制块 FCB . . . 282
7.3.4 文件操作 . . . 283
7.4 目录结构与路径名解析 . . . 283
7.4.1 目录文件 . . . 283
7.4.2 目录结构 . . . 283
7.4.3 加快目录检索 . . . 284
7.5 文件系统实现 . . . 284
7.5.1 磁盘与内存中的布局 . . . 285
7.5.2 文件的物理结构 . . . 286
7.5.3 UNIX 多级索引 . . . 288
7.5.4 磁盘空间管理 . . . 289
7.6 文件操作实现与共享 . . . 289
7.6.1 创建、打开、读写 . . . 289
7.6.2 rename 与 copy . . . 290
7.6.3 硬链接与软链接 . . . 291
7.7 文件系统实例 . . . 292
7.7.1 FAT . . . 292
7.7.2 UNIX 文件系统 . . . 292
7.7.3 NTFS . . . 292
7.8 XV6 文件系统代码阅读 . . . 292

7.9 本讲小结 . . . 295

第 8 讲：文件系统 2 . . . 297

8.1 本讲导读 . . . 297

8.2 文件系统可靠性 . . . 297

8.2.1 坏块问题 . . . 297

8.2.2 备份方式 . . . 298

8.3 文件系统一致性 . . . 298

8.3.1 为什么会不一致 . . . 298

8.3.2 `fsck` 检查 . . . 298

8.3.3 写入策略 . . . 299

8.4 文件系统安全性 . . . 299

8.4.1 身份认证 . . . 299

8.4.2 ACL 与 Capability . . . 300

8.4.3 RBAC 与 ABAC . . . 300

8.4.4 UNIX 权限 . . . 300

8.5 数据恢复 . . . 301

8.6 文件系统性能 . . . 301

8.6.1 块高速缓存 . . . 301

8.6.2 Windows 文件缓存 . . . 302

8.6.3 磁盘调度 . . . 303

8.6.4 RAID . . . 303

8.7 文件系统结构 . . . 304

8.7.1 VFS . . . 304

8.7.2 LFS 与日志文件系统 . . . 305

8.8 XV6 日志系统 . . . 305

8.9 分布式文件系统 . . . 306

8.9.1 Ceph . . . 306

8.9.2 HDFS . . . 307

8.10 本讲小结 . . . 307

第 9 讲：同步机制 . . . 310

9.1 本讲导读 . . . 310

9.2 开篇问题 . . . 310

9.3 并发与竞争条件 311
9.3.1 与时间有关的错误 311
9.3.2 线程参数传递竞态 312
9.4 互斥与临界区 312
9.4.1 基本概念 312
9.4.2 软件互斥算法 313
9.4.3 硬件方案 313
9.5 信号量与 P/V 操作 314
9.5.1 信号量定义 314
9.5.2 生产者-消费者 315
9.5.3 读者-写者 316
9.6 管程与条件变量 316
9.6.1 为什么引入管程 316
9.6.2 `wait` / `signal` 317
9.6.3 Hoare 与 Mesa 管程 318
9.7 Pthreads 同步 API 318
9.8 锁的实现与现代硬件 319
9.8.1 Futex 320
9.8.2 原子操作与内存顺序 320
9.9 XV6 并发同步机制 320
9.9.1 spinlock 与 sleeplock 320
9.9.2 sleep / wakeup 与 P/V 类比 321
9.9.3 XV6 pipe：生产者-消费者 322
9.10 附录：经典同步问题题型整理 322
9.10.1 考试解题模板 322
9.10.2 另类 P/V 操作问题 323
9.10.3 食品供货问题 324
9.10.4 三峡大坝船闸调度问题 325
9.10.5 狒狒过峡谷问题 327
9.10.6 睡眠理发师问题 329
9.10.7 复杂消息缓冲问题 331
9.10.8 写者优先与公平读者-写者 333
9.10.9 利用信号量管理共享资源 335

9.10.10 高频陷阱总结 338
9.11 本讲小结 339
第 10 讲：进程间通信 341
10.1 本讲导读 341
10.2 为什么需要 IPC 341
10.3 消息传递 342
10.3.1 基本接口 342
10.3.2 用 P/V 实现 send 343
10.3.3 用消息传递实现生产者-消费者 344
10.4 共享内存 344
10.5 管道、套接字与 RPC 345
10.5.1 管道 345
10.5.2 套接字 345
10.5.3 RPC 346
10.6 信号 347
10.7 Linux 内核同步机制 347
10.7.1 原子操作 347
10.7.2 自旋锁 347
10.7.3 自旋锁与信号量 349
10.7.4 读写锁 349
10.7.5 RCU 349
10.7.6 完成变量 350
10.7.7 屏障 351
10.8 本讲小结 351
第 11 讲：死锁 354
11.1 本讲导读 354
11.2 基本概念 354
11.2.1 死锁 354
11.2.2 活锁与饥饿 355
11.2.3 信号处理中的死锁 356
11.3 死锁产生的必要条件 356
11.4 资源分配图 357

11.4.1 图化简 . . . 357
11.5 死锁处理策略 . . . 358
11.5.1 鸵鸟算法 . . . 358
11.5.2 死锁预防 . . . 358
11.6 死锁避免与银行家算法 . . . 359
11.6.1 安全状态 . . . 359
11.6.2 银行家算法 . . . 360
11.6.3 银行家算法题 . . . 361
11.7 死锁检测与解除 . . . 362
11.8 哲学家就餐问题 . . . 362
11.8.1 朴素方案 . . . 362
11.8.2 管程方案 . . . 363
11.8.3 状态数组方案 . . . 364
11.8.4 限制人数方案 . . . 364
11.9 思考题整理 . . . 365
11.10 本讲小结 . . . 366
第 12 讲：I/O 管理 . . . 368
12.1 本讲导读 . . . 368
12.2 开篇问题 . . . 368
12.3 I/O 管理概述 . . . 369
12.3.1 I/O 的特点 . . . 369
12.3.2 设备发展 . . . 370
12.4 I/O 系统结构与设备分类 . . . 370
12.4.1 设备控制器 . . . 370
12.4.2 按数据特征分类 . . . 371
12.4.3 按资源分配分类 . . . 371
12.5 I/O 管理目标和任务 . . . 371
12.6 设备驱动模型 . . . 372
12.7 I/O 硬件组成 . . . 373
12.7.1 控制器接口 . . . 374
12.7.2 I/O 地址空间 . . . 375
12.8 I/O 控制方式 . . . 375

12.8.1 程序查询 I/O . . . 375
12.8.2 中断驱动 I/O . . . 375
12.8.3 DMA . . . 376
12.9 I/O 软件组成 . . . 376
12.9.1 设备命名和保护 . . . 376
12.10 缓冲技术 . . . 377
12.10.1 为什么需要缓冲 . . . 377
12.10.2 缓冲分类 . . . 378
12.10.3 缓冲控制块 . . . 378
12.11 设备分配 . . . 379
12.12 设备驱动程序与 I/O 进程 . . . 379
12.13 I/O 性能问题 . . . 381
12.13.1 同步、异步与阻塞、非阻塞 . . . 381
12.13.2 异步 I/O . . . 382
12.14 本讲小结 . . . 382

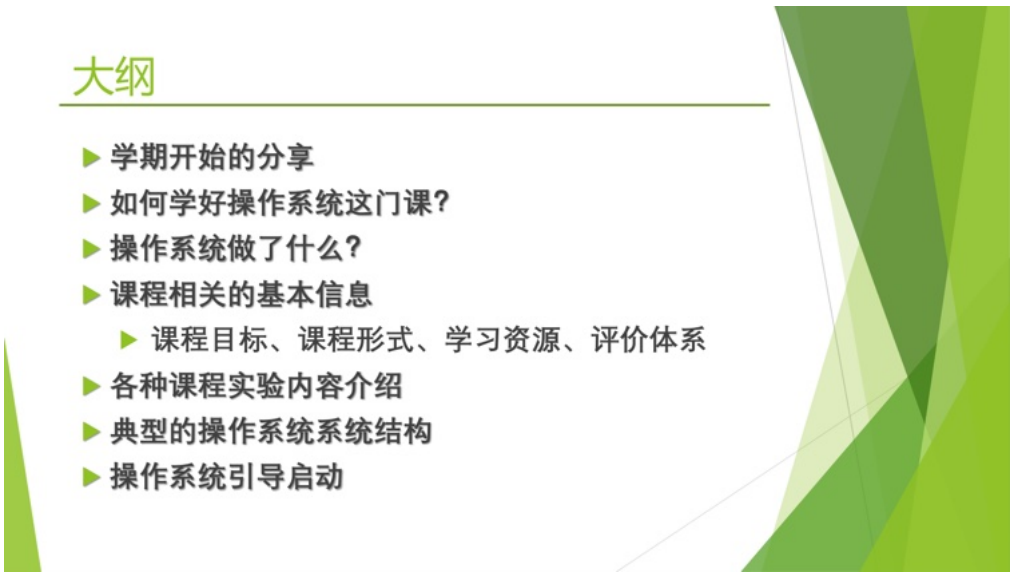
LECTURE 0

第 0 讲

课程简介

Operating Systems Course Notes

0.1 课程定位与学习方法



图示：课程大纲，展示课程简介、学习方法、实验体系和启动流程。

- **操作系统（Operating System）**课程不是“操作系统使用课”，而是理解操作系统如何工作的系统课程。
- 课程目标是把零散概念组织成一条主线：**基本概念 -> 典型流程 -> 关键技术 -> 实验验证 -> 复盘抽象**。
- 本课程强调：
  - 用系统性思维理解复杂软件系统；
  - 用源码阅读和实验把抽象机制落到代码；
  - 用典型执行流程把进程、内存、文件系统、I/O 串联起来。

0.2 如何学好操作系统

- 学习时不要只背定义，而要问：
  - 这个机制解决什么问题?
  - 它依赖什么硬件支持?
  - 它在内核代码中对应哪些数据结构?
  - 它和其他章节有什么联系?
- 建议维护几条贯穿全课的流程线：
  - 执行 HelloWorld 程序的流程；
  - `fork()` / `exec()` 的处理流程；
  - Page Fault 的处理流程；
  - 打开文件与一次 I/O 请求完成的流程。

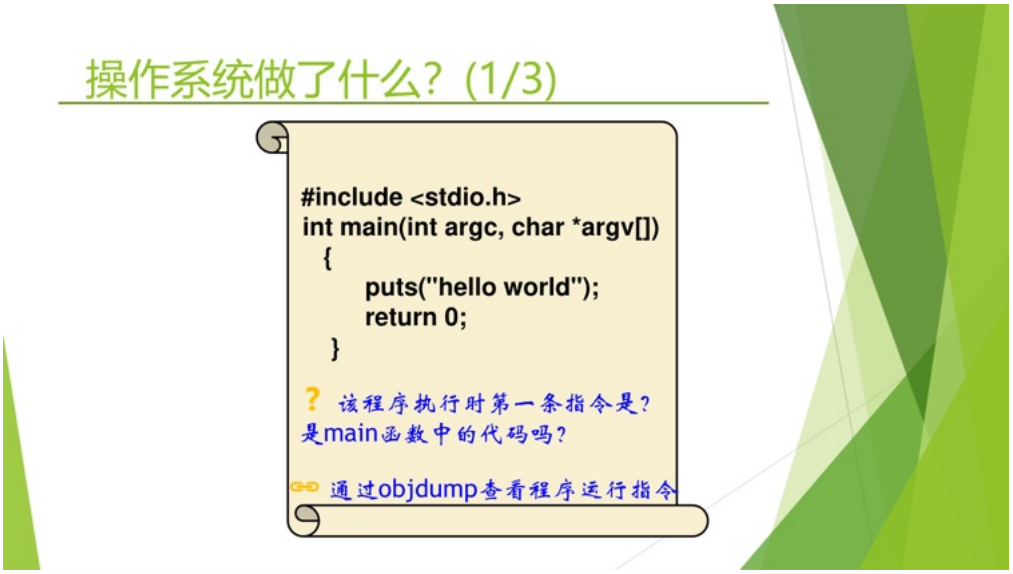


图示：学习操作系统应围绕基本概念、典型流程和关键技术。

> **思考题：**为什么学习操作系统要强调“构造性的思维活动”?

解答：操作系统是多个机制协作形成的系统工程。只被动接收概念，容易把进程、内存、文件系统、中断等内容割裂开；构造性思维要求我们主动建立因果链和执行链路，把知识组织成可解释、可推演的系统模型。

0.3 HelloWorld 背后的操作系统机制

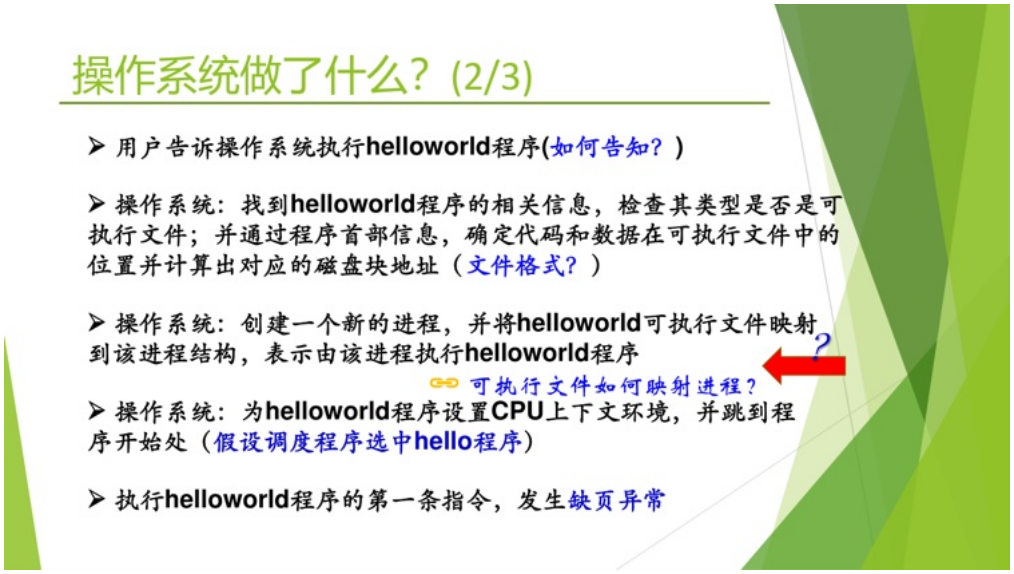


图示：HelloWorld 程序引出“第一条指令是否是 main”的问题。

> **思考题：** `HelloWorld` 程序执行时，第一条指令是 `main` 函数中的代码吗?

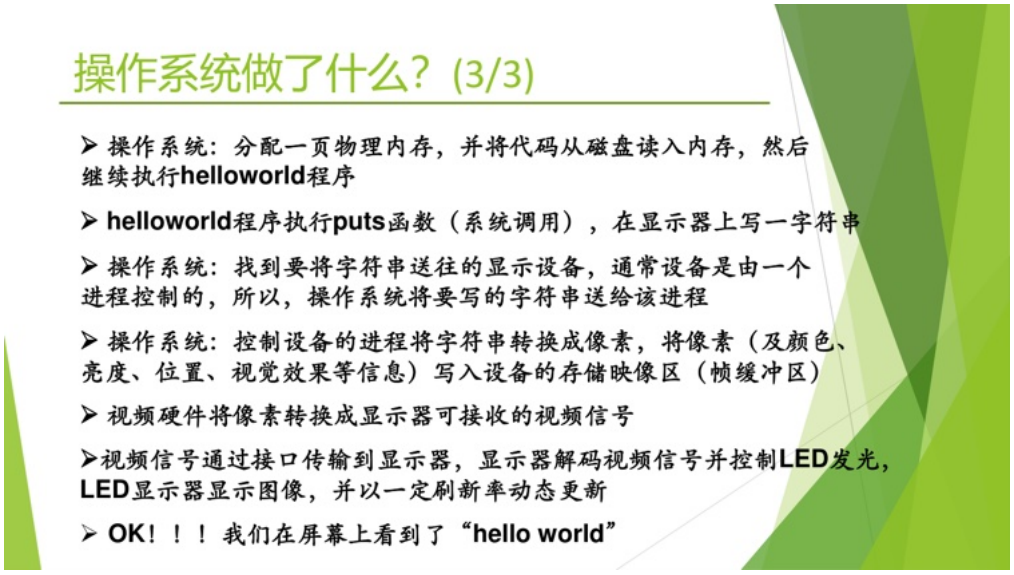
解答：不是。用户执行程序后，操作系统先解析可执行文件格式，创建进程，建立地址空间映射，设置初始用户态上下文；程序还要经过运行时启动代码完成参数、环境变量和 C 运行库初始化，

之后才调用 `main`。



图示：操作系统装载并启动 HelloWorld 的前半段流程。

- 典型流程：
  1. 用户通过 Shell 或图形界面发起执行请求；
  2. 内核找到可执行文件，检查文件格式；
  3. 内核根据 ELF 程序头确定代码段、数据段和入口地址；
  4. 创建进程控制块并建立地址空间；
  5. 设置 CPU 上下文，等待调度运行。



图示：缺页、系统调用和显示输出。

- HelloWorld 涉及的模块：
  - **进程管理**：创建进程、调度、上下文切换；
  - **内存管理**：地址空间、页表、缺页异常；

- **文件系统**：读取可执行文件；
- **I/O 管理**：系统调用、设备驱动、显示输出。

0.4 课程实验与源码阅读



图示：XV6 实验覆盖系统调用、页表、陷入、线程、锁、文件系统等模块。

- 实验不是附属内容，而是理解操作系统机制的主通道。
- XV6 的价值在于规模小、结构清晰，可以把课堂机制对应到真实内核源码。
- 阅读 XV6 时应关注：
  - 入口路径：从 trap/syscall 到具体内核函数；
  - 核心数据结构：`proc`、`trapframe`、`pagetable`、`inode`；
  - 状态变化：进程状态、页表项、文件引用计数等。

0.5 启动流程与课程主线

> **代码阅读题：**总结 XV6-RISCV 的多核启动流程。

解答： XV6-RISCV 的启动通常从 `_entry` 开始，为每个 hart 准备独立内核栈；随后进入 `start`，完成机器态到监管态的必要设置，如设置 `mepc`、委托中断异常、配置时钟中断等；执行 `mret` 后进入 `main`。`main` 中只有一个核心负责初始化内核全局资源，其他核心等待初始化完成后进入调度循环。这体现了多核启动中“单核初始化共享资源，多核随后并发运行”的基本原则。

0.5.1 从固件到内核：启动链路的分层理解

- **固件阶段**：
  - 机器上电后最先运行的是固件，而不是操作系统内核。
  - 固件完成基本硬件初始化，并找到可启动设备。



图示：XV6-RISCV 启动链路。

  - 在 PC 体系中常见固件包括 BIOS 和 UEFI；在 RISC-V 教学环境中通常由模拟器或启动代码提供初始入口。
- **引导程序阶段**：
  - 引导程序负责把内核镜像从存储介质加载到内存。
  - 它需要理解磁盘布局、内核文件格式和内存装载地址。
  - 它的职责不是提供完整服务，而是“把内核带到能够开始运行的位置”。
- **内核入口阶段**：
  - 内核入口代码通常用汇编书写，因为此时 C 语言运行环境尚未完全建立。
  - 入口代码要准备栈、关键信息和最小运行环境。
  - 随后跳转到 C 语言写成的初始化函数。
- **内核初始化阶段**：
  - 初始化物理内存管理、页表、进程表、中断/异常、设备和文件系统。
  - 创建第一个用户进程或进入调度循环。
  - 多核系统还要处理不同 CPU 核心之间的启动顺序和同步。

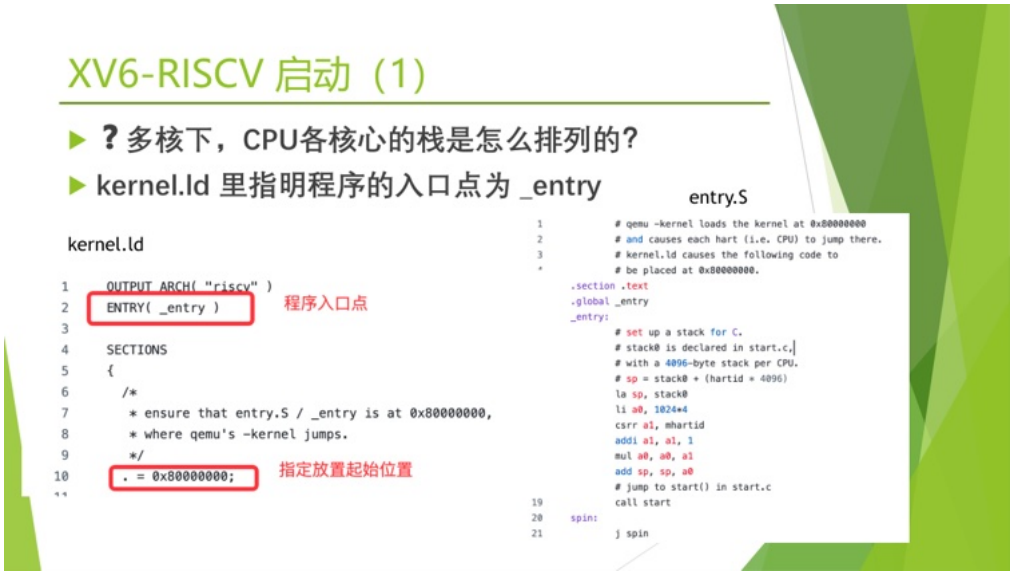
> **重点：**启动不是单个函数调用，而是一条从硬件控制权逐步转移到内核控制权的链路。

0.5.2 多核启动中的栈安排

> **思考题：**多核下，CPU 各核心的栈是怎么排列的?

解答：

- 每个 CPU 核心都需要独立的内核栈。
- 原因是不同核心可能同时进入内核，如果共享同一个栈，会互相覆盖返回地址、局部变量和保



图示：多核启动中栈空间和入口代码的安排。

存的寄存器。

- 常见做法是在内核的数据区中预留一段连续栈空间：
  - 每个核心根据自己的 hart id / CPU id 计算栈顶；
  - 栈一般向低地址增长；
  - 不同核心的栈区域互不重叠。
- 这样每个核心进入内核后都能安全地执行函数调用和中断处理。

可以把它理解为：

```
CPU0 -> stack0
CPU1 -> stack1
CPU2 -> stack2
...
```

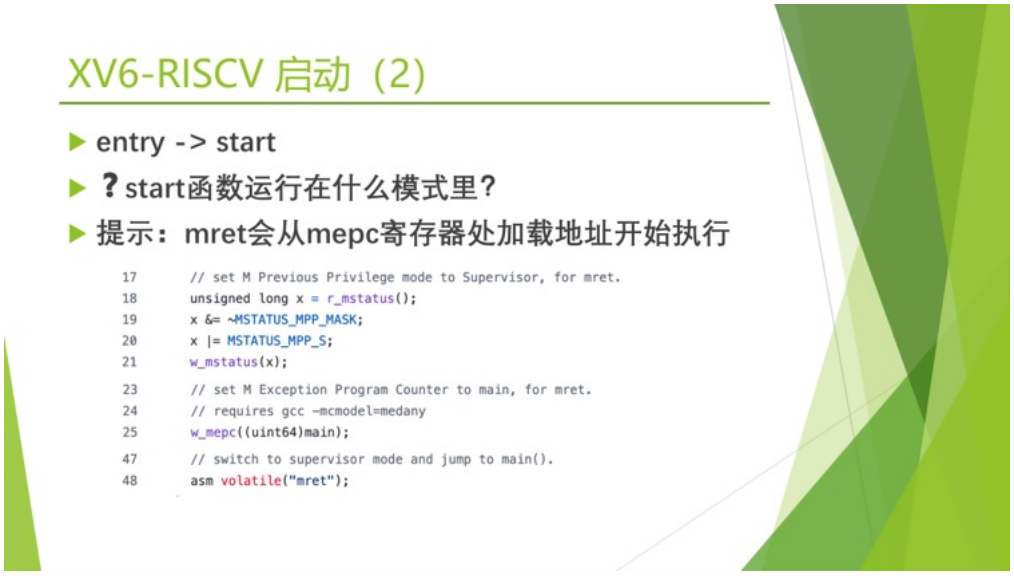
每个核心都从同一份内核代码开始执行，但必须使用不同的运行时栈。

0.5.3 `start` 运行在什么模式里？

> **思考题：** `start` 函数运行在什么模式里？为什么 `mret` 会从 `mepc` 寄存器处加载地址开始执行？

解答：

- RISC-V 机器启动后通常先处于 **机器态（Machine Mode, M-mode）**。
- `start` 阶段的任务是完成从机器态到监管态的准备：
  - 设置 `mepc`，指定 `mret` 返回后要执行的地址；
  - 设置 `mstatus`，指定返回后的特权级；
  - 配置中断委托，使部分中断/异常交给监管态处理；



图示：从入口代码进入 start 函数的过程。

  - 初始化时钟中断相关寄存器。
- `mret` 是从机器态返回到较低特权级的指令。
- 执行 `mret` 时，处理器会读取 `mepc` 中保存的地址，并从该地址继续执行。

因此，`start` 的核心意义是：**把硬件启动后的最高特权控制权，安全地交给操作系统内核运行环境**。



图示：从 start 进入 main 后完成内核初始化。

0.5.4 从 `main` 看内核初始化

> **代码阅读题：**为什么多核启动时通常只有一个核心负责主要初始化？

解答：

- 内核初始化会修改许多全局数据结构：
  - 空闲物理页链表；
  - 进程表；
  - 文件表；
  - 缓冲区缓存；
  - 设备驱动状态；
  - 中断控制器配置。
- 如果多个核心同时初始化这些结构，会产生竞态条件。
- 因此常见方案是：
  1. 让一个主核心完成全局初始化；
  2. 设置一个全局标志表示初始化完成；
  3. 其他核心等待该标志；
  4. 初始化完成后，所有核心进入调度循环。

这个模式体现了操作系统中常见的原则：

- **共享资源初始化必须串行化**；
- **并发运行之前先建立同步基础**；
- **多核并发依赖锁、中断和调度器等基础设施**。

0.6 实验体系与工程能力

0.6.1 为什么要做操作系统实验

- 操作系统实验的目的不是“完成作业”，而是把抽象机制落到可运行代码中。
- 许多机制只有在实现时才会真正暴露复杂性：
  - 系统调用要考虑参数传递、权限检查和返回路径；
  - 页表要考虑地址翻译、权限位和缺页处理；
  - 调度要考虑进程状态、锁和中断；
  - 文件系统要考虑缓存、一致性和崩溃恢复。

> **总结：**实验让“概念”变成“可执行机制”，也让你看到设计权衡如何体现在代码中。

0.6.2 XV6 为什么适合作为学习材料

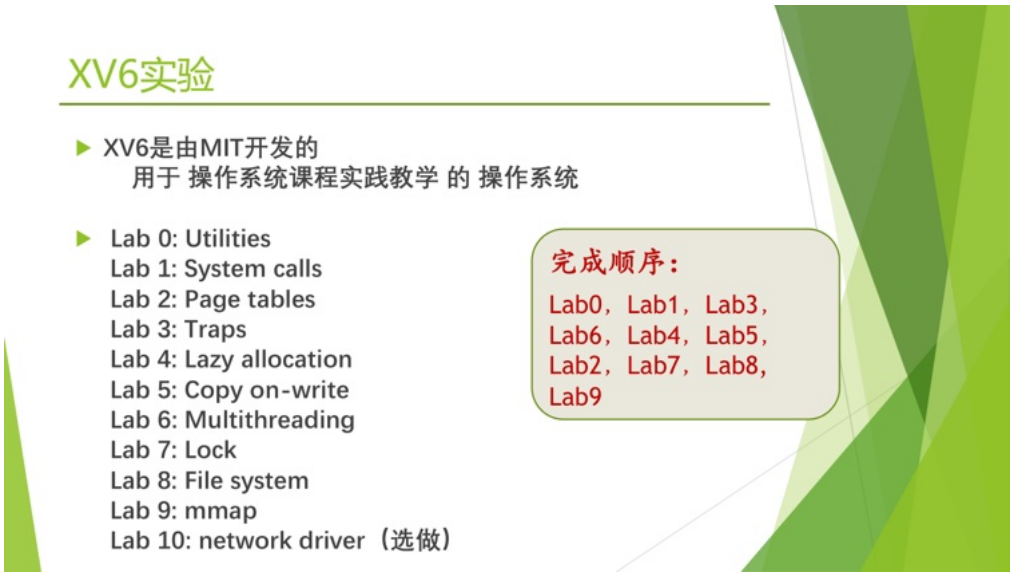
- **规模适中**：代码量远小于真实 Linux 内核，适合课程中完整阅读。
- **机制完整**：包含进程、内存、系统调用、中断、文件系统、锁等核心模块。
- **结构清晰**：模块边界相对清楚，便于把课堂概念映射到源码。
- **硬件相关性适中**：既能体现真实硬件机制，又不会被大量工程细节淹没。



图示：可选实验路线与教学操作系统。

常见实验模块包括：

模块	训练目标
Utilities	熟悉用户态工具和系统调用使用
System calls	理解用户态到内核态的受控入口
Page tables	理解虚拟地址、页表和权限
Traps	理解中断、异常和系统调用统一入口
Lazy allocation	理解按需分配和缺页处理
Copy-on-write	理解 fork 优化和页面共享
Multithreading	理解线程与并发控制
Lock	理解同步和竞态
File system	理解 inode、目录和块缓存
mmap	理解文件映射和虚拟内存结合



图示：XV6 实验模块和对应机制。

0.6.3 做实验时应该记录什么

- **问题背景**：当前实验要解决哪个操作系统问题？
- **关键数据结构**：新增或修改了哪些结构体字段？
- **控制流**：从用户调用到内核处理再到返回的路径是什么？
- **同步关系**：是否涉及锁、中断关闭、引用计数？
- **测试方法**：如何证明代码正确？
- **反思总结**：实现中遇到的困难来自概念理解、代码结构还是边界条件？

这些记录会帮助你把实验从“写过代码”提升为“真正理解机制”。

0.7 本讲总结

0.7.1 需要带走的主线

- 操作系统是系统工程，不是知识点列表。
- HelloWorld 的执行路径能串联进程、内存、文件系统和 I/O。
- 实验和源码阅读是理解机制的核心方式。
- 启动流程体现了从硬件到内核、从单核初始化到多核并发的控制权转移。

0.7.2 复习检查表

- ☐ 能解释 `main` 为什么不是程序执行的第一条指令。
- ☐ 能说出 HelloWorld 至少涉及哪些内核子系统。
- ☐ 能画出从固件、引导程序到内核入口的启动链路。
- ☐ 能说明多核启动为什么需要每个核心独立内核栈。
- ☐ 能说明 XV6 实验和课程章节的对应关系。

LECTURE 1

第 1 讲

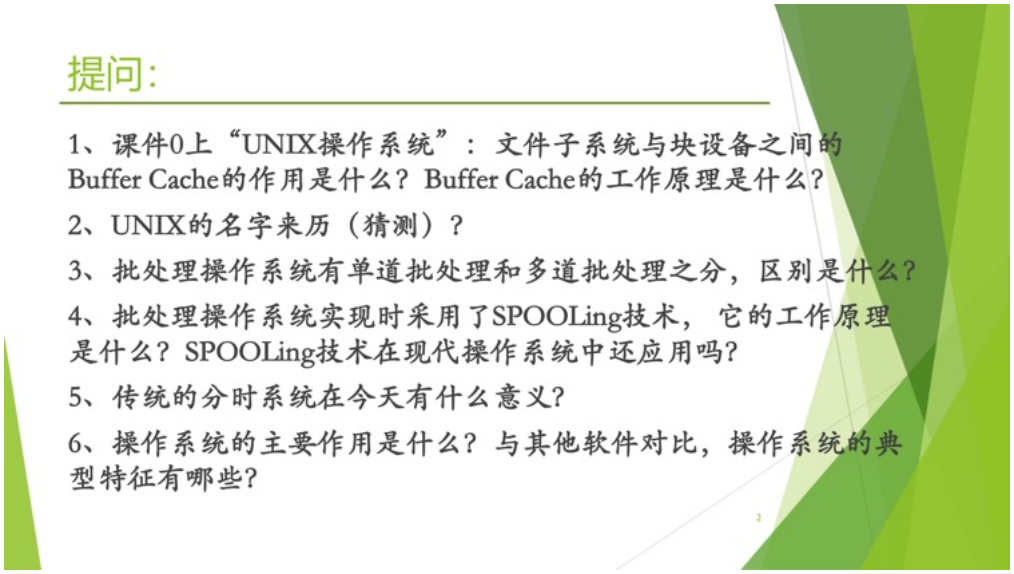
操作系统概述

Operating Systems Course Notes

1.1 本讲导读

- 这一讲围绕一个根本问题展开：在裸硬件之上，**为什么必须存在操作系统**，以及它究竟在做什么。
- 知识脉络：定义 -> 三种认知视角 -> 四个基本特征 -> 发展阶段与启示 -> 操作系统分类。
- 学习时不要把它当成定义的堆叠。每一段都应回答“它解决了什么问题、是怎么实现的、留下了什么后果”。

1.2 开篇问题



图示：本讲开篇提出的 6 个问题。这些问题是后面所有内容的引线。

> **思考题（开篇问题 1）：** UNIX 文件子系统与块设备之间的 Buffer Cache 的作用是什么？工作原理是什么？

解答：

- **位置与角色**：Buffer Cache 位于文件系统逻辑层和块设备驱动之间，是一段由内核维护的内存缓冲池，每个缓冲区对应一个磁盘块。
- **核心目的**：磁盘 I/O 远慢于 CPU 与内存。如果每次文件读写都触发磁盘访问，系统性能会严重下降。Buffer Cache 让常用块停留在内存中，减少磁盘访问次数。
- **读路径**：读一个磁盘块时，内核先在缓存中查找；命中则直接返回内存中的副本；未命中才发起磁盘 I/O，并把读入的数据放入缓存。
- **写路径**：写操作通常先写入缓存，并把对应缓冲区标记为“脏”。内核稍后按策略（如周期同步、容量压力下回写、`sync` 调用）把脏块写回磁盘。
- **替换策略**：缓存空间有限，一般使用近似 LRU 的策略淘汰最少使用的块。
- **设计代价**：缓存提高了性能，但带来一致性和崩溃恢复问题。系统崩溃时未写回的脏块会丢

失，所以现代文件系统会引入日志或同步刷盘策略。

> **思考题（开篇问题 2）：** UNIX 名字的来历?

解答：

- UNIX 的命名公认来源是对 MULTICS（MULTiplexed Information and Computing Service）的戏仿。
- MULTICS 强调“多路复用、面向多用户”，但工程上极度复杂；Ken Thompson 等人在 PDP-7 上做的小型替代系统反向选了“UNI”作为前缀，强调“单一、简洁”。
- 这个命名也体现了 UNIX 的设计哲学：把复杂功能拆成简单工具，再组合起来。

> **思考题（开篇问题 3 与 4）：** 单道批处理、多道批处理与 SPOOLing 分别解决什么问题?

解答：

机制	解决的问题	核心思想	代价
单道批处理	减少人工换卡、换带的时间	作业按批顺序自动执行	I/O 时 CPU 仍空闲
多道批处理	CPU 在 I/O 等待时被浪费	多道作业同时驻留内存，I/O 阻塞时切换执行	需要内存保护、调度、资源管理
SPOOLing	慢速独占设备拖慢整个系统	用磁盘做缓冲，把输入、计算、输出解耦	需要排队、缓冲区与守护进程管理

SPOOLing 的“缓冲 + 排队 + 异步”思想至今仍在使用，比如打印队列、网络发送队列、日志缓冲区、消息中间件。

> **思考题（开篇问题 5）：** 传统分时系统在今天还有意义吗?

解答：

- 分时系统的本质是“在时间维度共享一台计算机”，让多个交互用户都能获得快速响应。
- 现代多用户服务器、多核多任务桌面操作系统、容器和云上的多租户系统都直接继承了分时思想。
- 因此分时系统不是历史名词，而是当前操作系统调度模型的基础。

> **思考题（开篇问题 6）：** 操作系统的主要作用与典型特征?

解答： 这是后面 §2 与 §4 的内容，本节先把它作为索引留下。

1.3 操作系统的定义与作用



图示：从“资源管理者”“服务提供者”“机器能力的扩展”三个层面理解操作系统的定义。

1.3.1 一句话与三层含义

操作系统（Operating System）是计算机系统中的一种系统软件，是一组协同工作的程序模块。它同时承担三种角色：

- **资源的管理者**：以有效、合理的方式组织、跟踪、分配和回收计算机的软硬件资源。
- **服务的提供者**：组织计算机的工作流程，控制程序的执行，并为应用提供可调用的接口和标准库。
- **机器能力的扩展**：把裸硬件包装成更易编程、更可控的“扩展机器”。

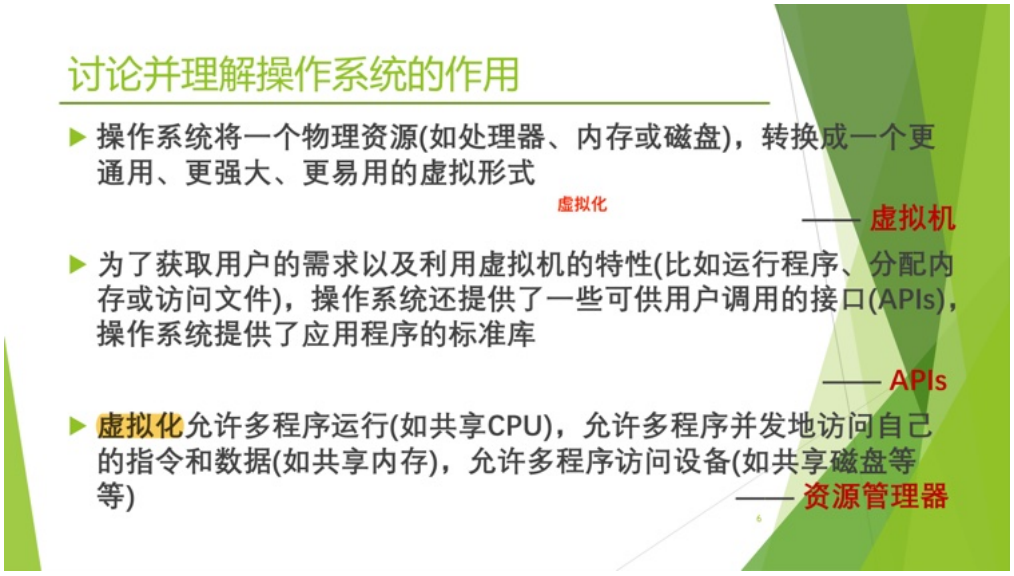
这三种角色不是并列的功能列表，而是同一组代码同时呈现出的三个侧面。

1.3.2 三种作用：虚拟机 / API / 资源管理器

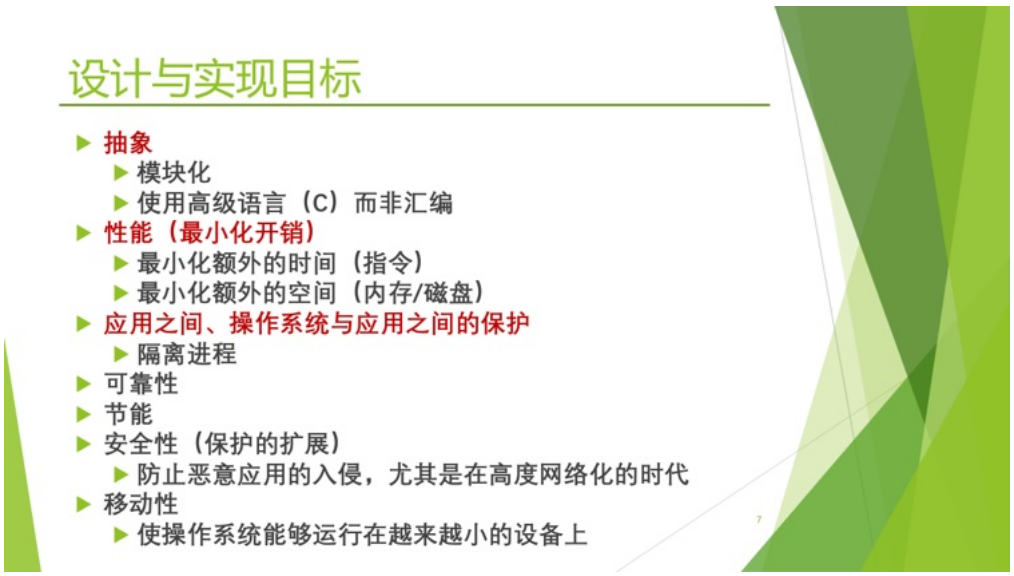
- **虚拟机**：把物理 CPU、内存、磁盘转换为更通用、更易用的虚拟形式。例如多个进程都觉得自己独占 CPU 与内存。
- **API**：通过系统调用把虚拟资源能力暴露给应用，应用不必直接面对硬件细节。
- **资源管理器**：当多个应用共享同一硬件时，操作系统决定“谁、何时、用多少”，并在它们之间防止互相破坏。

1.3.3 设计与实现目标

- **抽象**：通过分模块设计与高级语言（典型是 C）实现，避免被汇编绑死在具体硬件平台。
- **性能**：尽量减少额外的指令开销与额外的内存/磁盘开销。
- **保护**：在应用与应用、应用与内核之间建立隔离边界，使用硬件机制（特权级、内存保护）共同实现。
- **可靠性 / 节能 / 安全 / 移动性**：随时代演进而被加进设计目标，体现了操作系统不是静态



图示：操作系统把物理资源转化为虚拟资源，并通过 API 把虚拟资源暴露给应用。



图示：操作系统在抽象、性能、保护、可靠性、节能、安全、移动性等维度的目标。

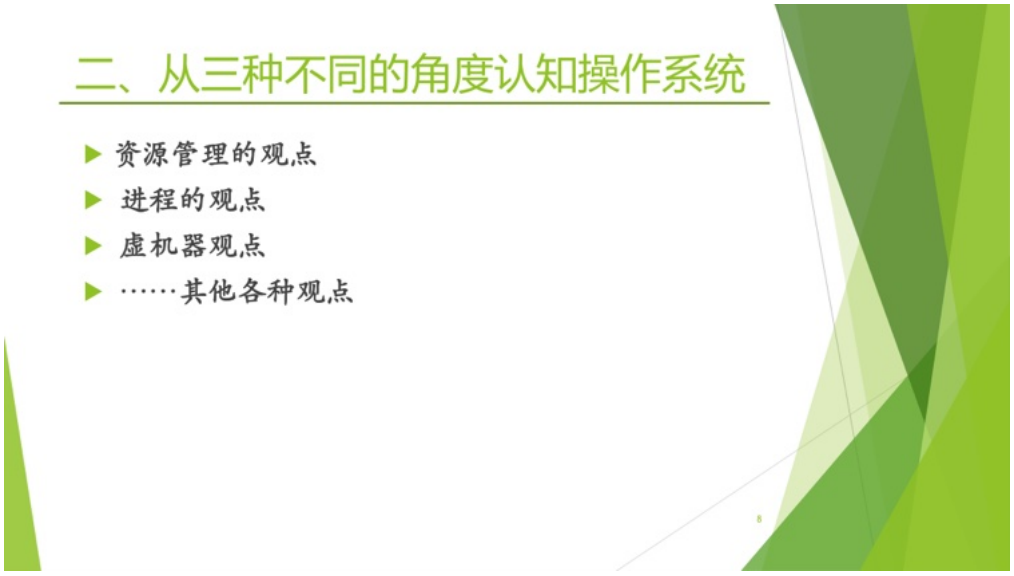
产品，而是与硬件、应用、网络环境一起演化。

> 一个反复出现的工程权衡：保护越强、抽象越多，性能开销越高。操作系统的发展史就是不断在“更安全易用”和“尽量不损失性能”之间找折中。

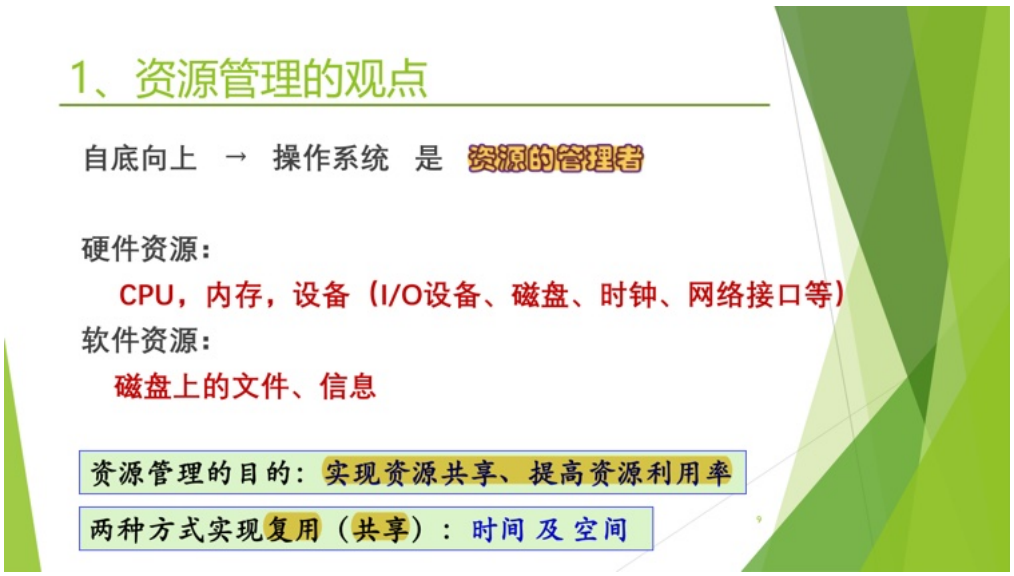
1.4 三种认知视角

1.4.1 资源管理视角（自底向上看）

- **硬件资源**：CPU、内存、I/O 设备、磁盘、时钟、网络接口等。
- **软件资源**：磁盘上的文件、目录与各种数据。
- **管理目标**：实现共享、提高利用率。



图示：资源管理视角、进程视角和虚机器视角，是认识操作系统的三条主要切入路径。

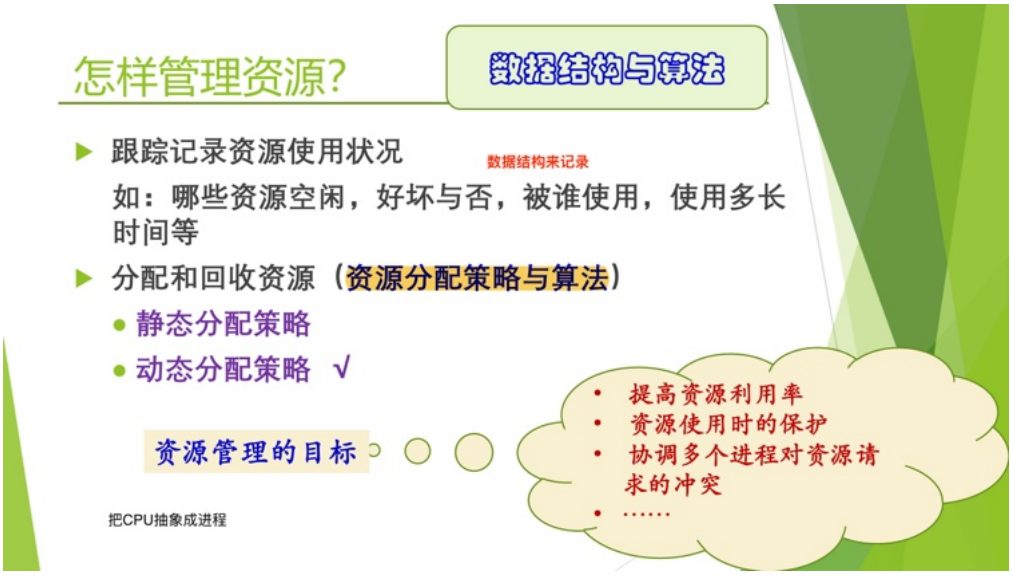


图示：操作系统作为资源管理者，管理硬件资源（CPU、内存、设备）和软件资源（文件、信息）。

- **复用方式**：
  - **时间复用**：让不同请求在不同时间使用同一资源（如 CPU 时间片轮转）。
  - **空间复用**：让多个请求在不同区域同时使用同一资源（如内存分区、磁盘块分配）。

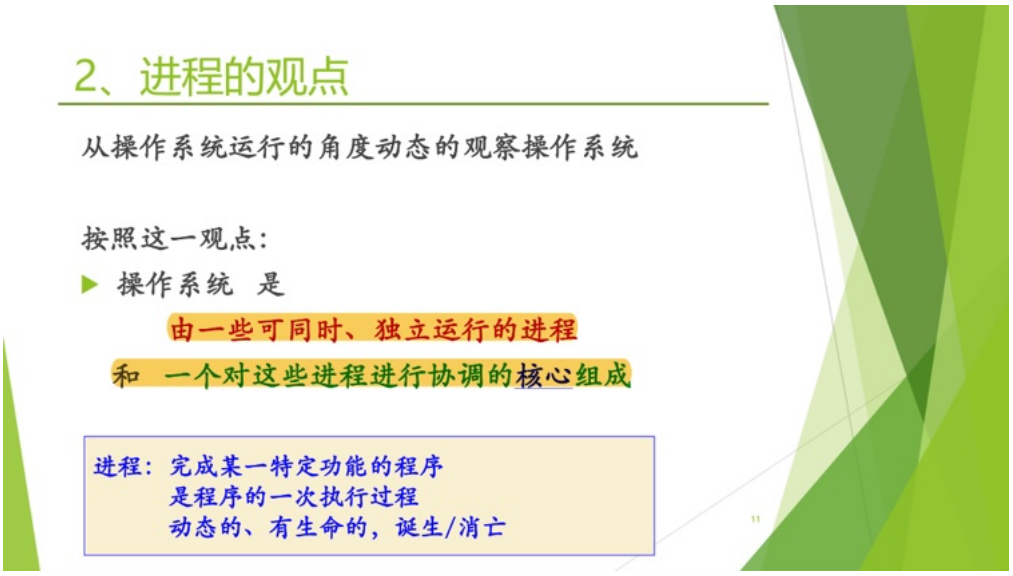
操作系统通过两类工具管理资源：

- **数据结构**：记录“哪些资源空闲、被谁使用、使用多久”。
- **算法**：执行分配与回收。
  - 静态分配：运行前一次性分配，简单但僵硬。
  - 动态分配：按需申请释放，灵活但复杂。
- **管理目标**：提高利用率、保护资源、协调请求冲突。



图示：用数据结构记录资源使用，用算法实现分配与回收。

1.4.2 进程视角（动态看）

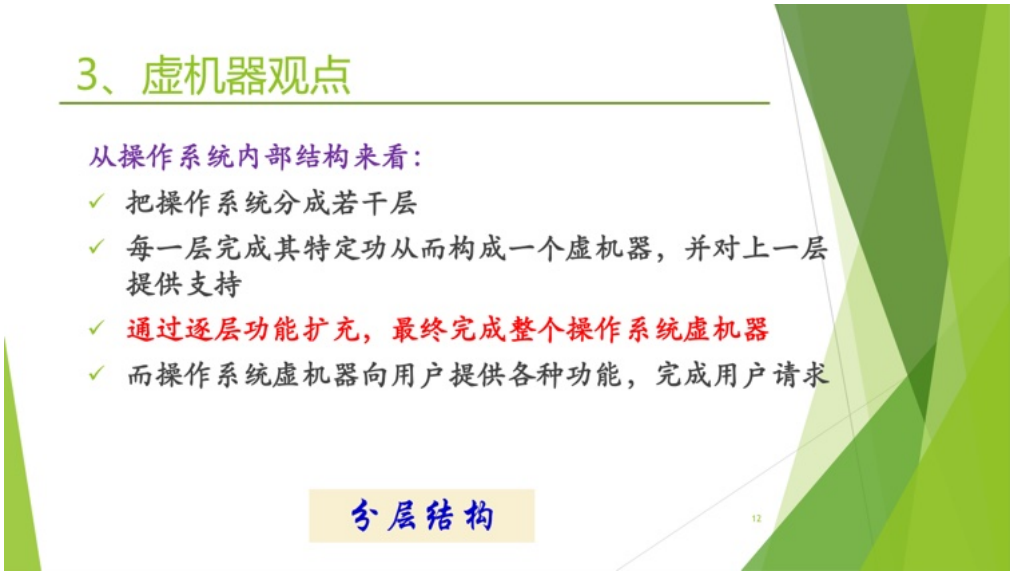


图示：从运行过程看，操作系统是一组协同执行的进程加上协调它们的核心。

- 这一视角把注意力放在“运行时是什么在动”。
- **进程（Process）**是某段程序的一次执行过程，是动态的、有生命周期的实体。
- 操作系统由若干并行进程加上一个协调内核组成。这一视角直接通向第 3 讲的进程线程模型。

1.4.3 虚机器视角（结构看）

- 每一层完成自己的功能，构成一个“虚拟机器”。
- 上层基于下层提供的接口工作，不需要关心更底层细节。
- 这种分层既是组织方式，也是教学上理解操作系统模块关系的工具。

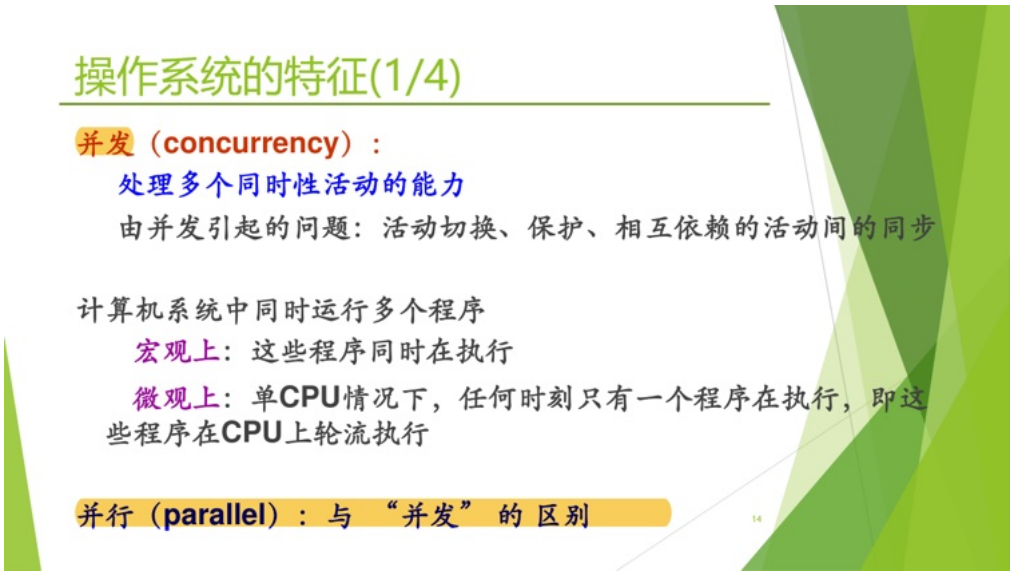


图示：通过分层组织把操作系统看成一组堆叠的虚拟机器，每一层为上一层提供服务。

1.5 操作系统的特征

操作系统区别于普通应用软件，体现在四个并列的特征上：并发、共享、虚拟、随机性。

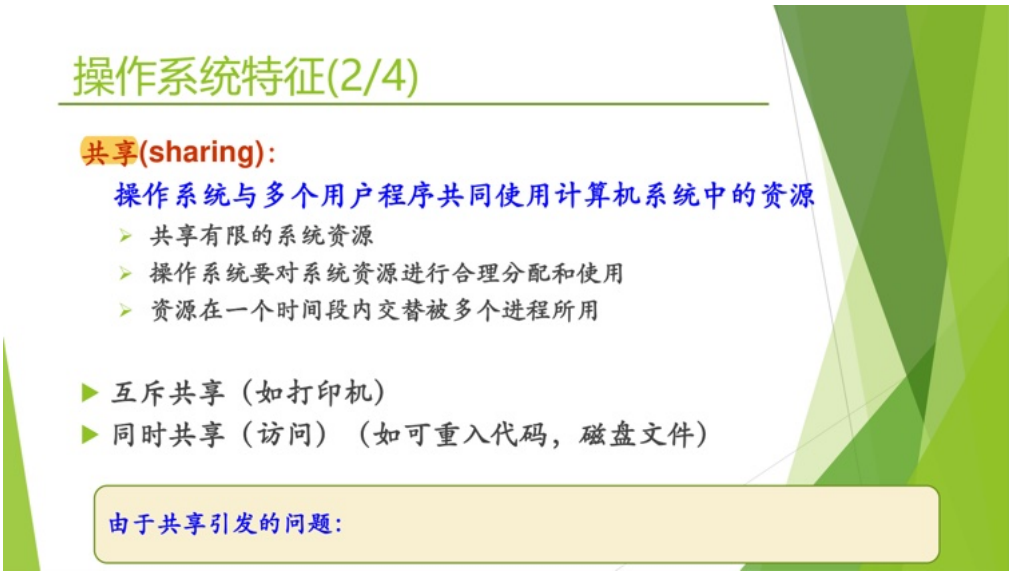
1.5.1 并发（Concurrency）



图示：宏观同时、微观交替，是并发的本质。

- **并发**是处理多个同时性活动的能力。
- **宏观上**：多个程序“同时”运行。
- **微观上**：单 CPU 上一次只跑一个程序，多程序在 CPU 上轮流执行。
- **并行（Parallel）**与并发不同。并行强调真正同时执行，需要多核或多处理器；并发强调能否处理多个任务的交错。
- 并发带来的问题：上下文切换、保护、相互依赖活动间的同步。

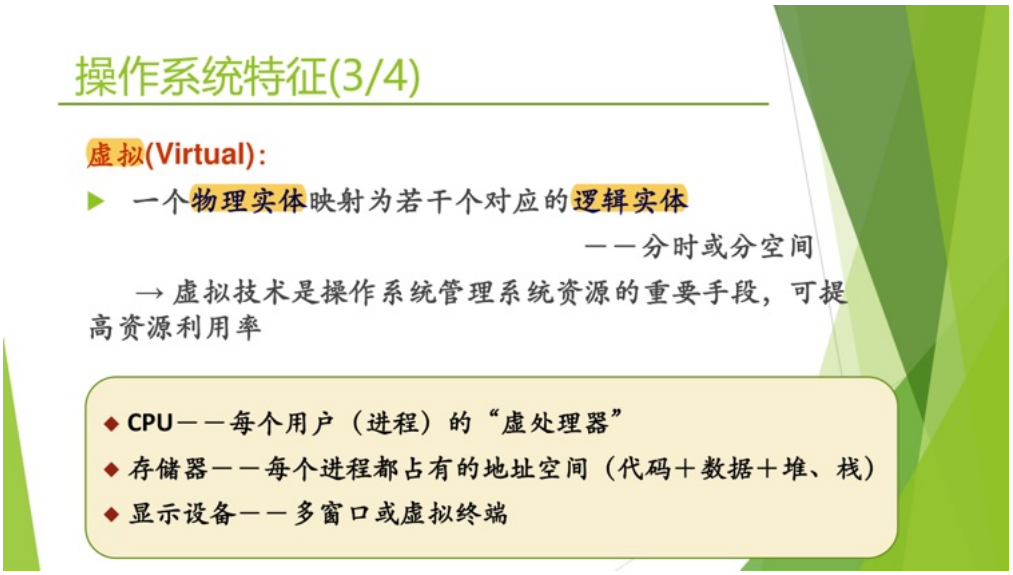
1.5.2 共享（Sharing）



图示：互斥共享与同时共享。

- 操作系统协调多个用户/进程共同使用有限的系统资源。
- **互斥共享**：一段时间内只能由一个进程独占使用，例如打印机。
- **同时共享**：可以同时被多个进程访问，例如可重入代码、磁盘上同一文件的只读访问。
- 共享带来的问题：资源分配难以最优、使用时必须保护、容易触发竞态条件。

1.5.3 虚拟（Virtual）

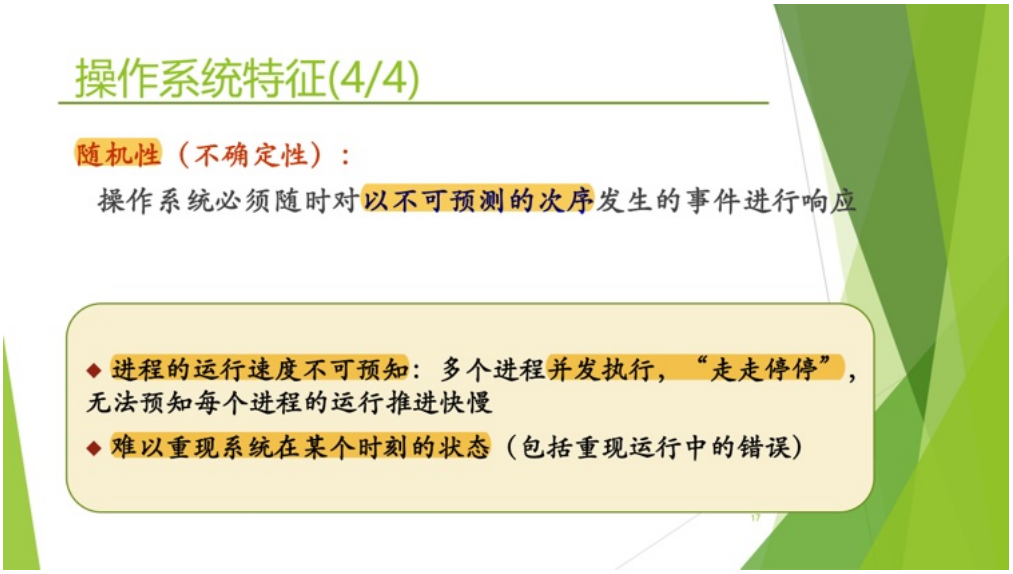


图示：把一个物理实体映射成若干个逻辑实体（分时或分空间）。

- **虚拟**的本质是“一个物理实体被映射成若干个对应的逻辑实体”。
- 实现手段：分时（如 CPU 给每个进程虚拟出独占的处理器）和分空间（如把一台机器的内存按地址空间切分给多个进程）。

- 典型对应：
  - CPU -> 进程的虚处理器
  - 内存 -> 各进程独立的地址空间
  - 显示设备 -> 多窗口、虚拟终端

1.5.4 随机性（不确定性）



图示：操作系统必须随时响应不可预测顺序到达的事件。

- 进程的运行速度不可预测，因为受到调度、I/O、外部中断等多种因素影响。
- 同样的程序在系统某一时刻的状态也难以重现，调试操作系统因此格外困难。
- 这一特征意味着：内核代码必须假设“任何时间都可能被打断”，并依此设计同步机制。

1.6 发展历程及启示

操作系统的发展不是单线进化，而是“硬件越便宜、人工越贵 -> 系统形态调整”的反复过程。课件按 7 个阶段梳理了这条线索。

1.6.1 第 1 阶段（1948–1970）：硬件昂贵，人工便宜

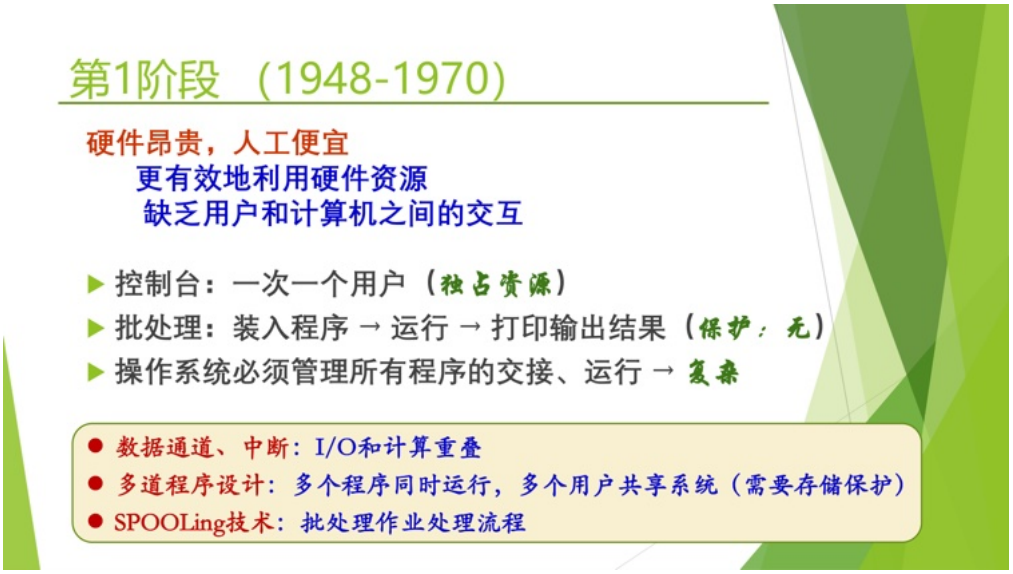
- 计算机昂贵，必须充分利用硬件资源。
- 演进路径：控制台独占 -> 单道批处理 -> 多道批处理 -> 引入存储保护、I/O 通道、SPOOLing。

1.6.2 第 2 阶段（1970–1985）：硬件便宜，人工昂贵

- 计算资源相对便宜后，开始牺牲一些 CPU 时间换取交互响应。
- 分时系统先在 MIT CTSS、Manchester Atlas 等系统上落地，把“多用户共享一台计算机”变成现实。



图示：从大型机、PC、网络、移动计算到云计算与人机物融合的发展脉络。



图示：从控制台、单道批处理到多道程序设计与 SPOOLing。

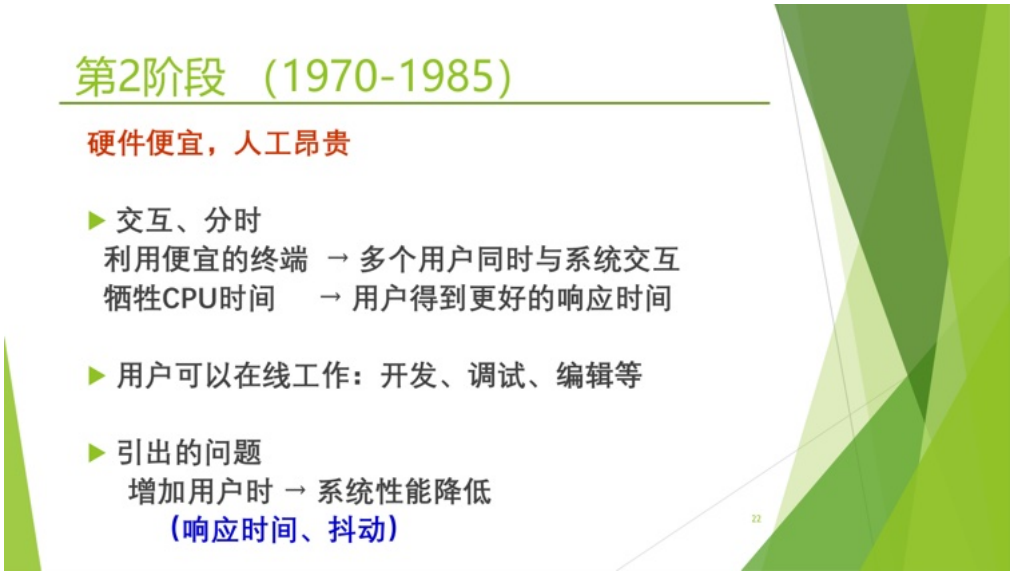
- 同期发生了 OS/360 的工程灾难和 MULTICS 的延期发布，揭示了大型操作系统在工程组织上的复杂性。
- UNIX 在贝尔实验室作为“小而美”的反面案例诞生，并最终深刻影响后来的所有主流操作系统。

1.6.3 第 3 阶段（1981 起）：个人计算时代

- PC 早期硬件资源有限，操作系统退化为简单的运行时库。
- 当 PC 的资源逐渐变得丰富，多任务、存储保护、虚拟内存等机制再次进入桌面操作系统。

1.6.4 第 4 阶段（1981 起）：分布式

- 网络化让多台机器之间共享资源成为常态。



图示：从批处理走向分时，提升用户响应。



图示：CTSS 的关键参数与早期分时实现细节。

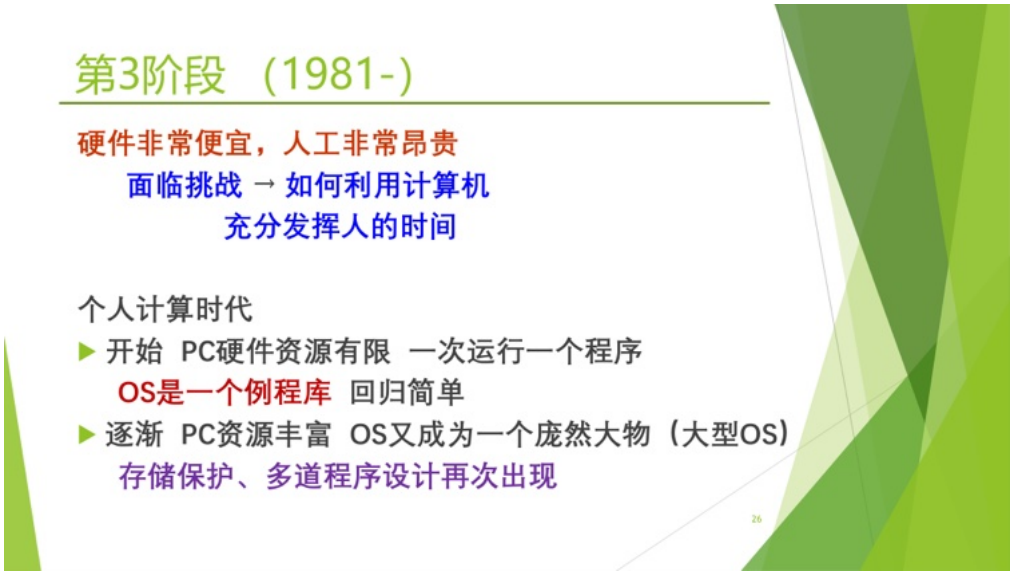
- 操作系统要解决跨机共享与跨机安全问题。

1.6.5 第 5 阶段（1995 起）：移动计算

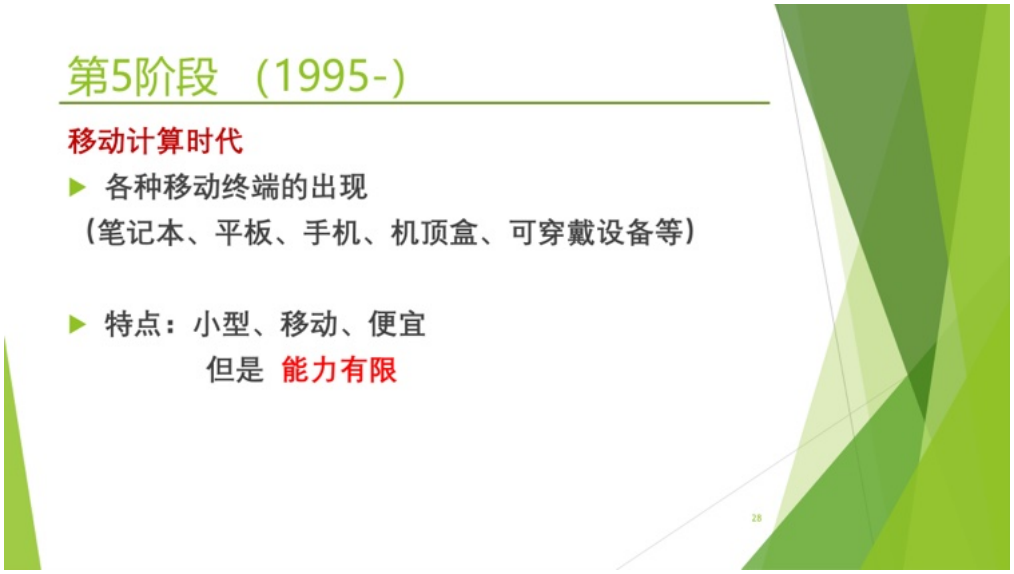
- 设备小型化、便携化，但能力受限。
- 移动操作系统在能耗管理、通信集成、用户体验等方向发展出新的子领域。

1.6.6 第 6 阶段（2006 起）：云计算

- 资源以可弹性伸缩、按需获取、按使用付费的方式提供。
- 数据中心层面的“云操作系统”负责管理海量服务器、虚拟化、分布式文件系统、调度和安全。



图示：PC 早期回归简单，随后再次发展为大型 OS。



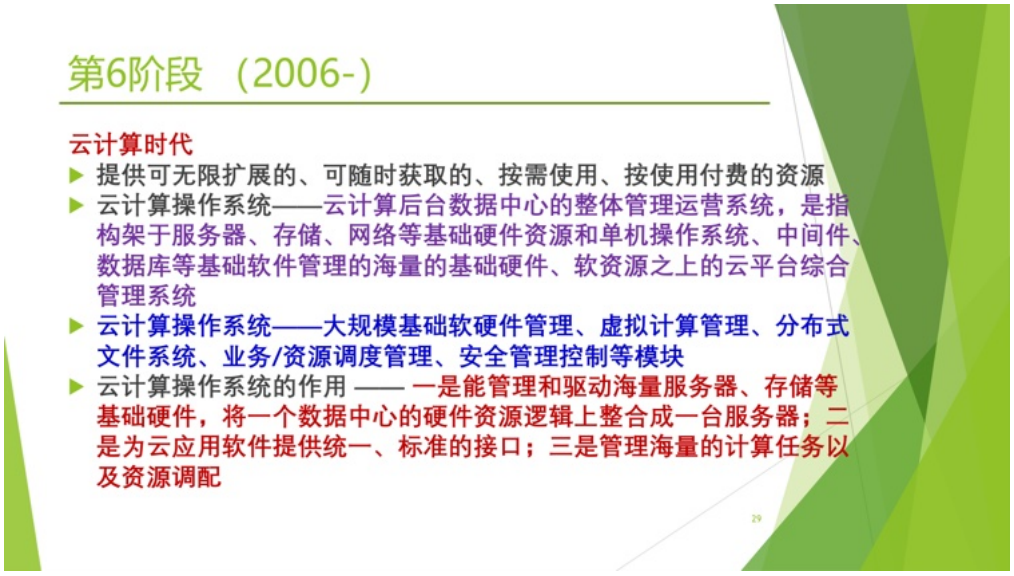
图示：手机、平板、可穿戴设备催生新一代移动操作系统。

1.6.7 第 7 阶段（2000 年代起）：泛在计算 / 物联网

- 大量联网设备协同，把计算嵌入物理世界。
- 操作系统的边界扩展到“人–机–物融合”的环境。

> **历史启示：**多数“老”概念都没有真正消失。SPOOLing 在打印队列里复活，分时思想在容器调度里再现，分布式概念在云计算中放大。学习操作系统时要看到“概念的回归”，而不是只看技术细节的更替。

1.7 操作系统分类



图示：数据中心规模的“操作系统”，管理硬件、虚拟化、调度与服务。



图示：批处理、分时、实时、个人计算机、网络、分布式、嵌入式等主要类别。

1.7.1 批处理操作系统

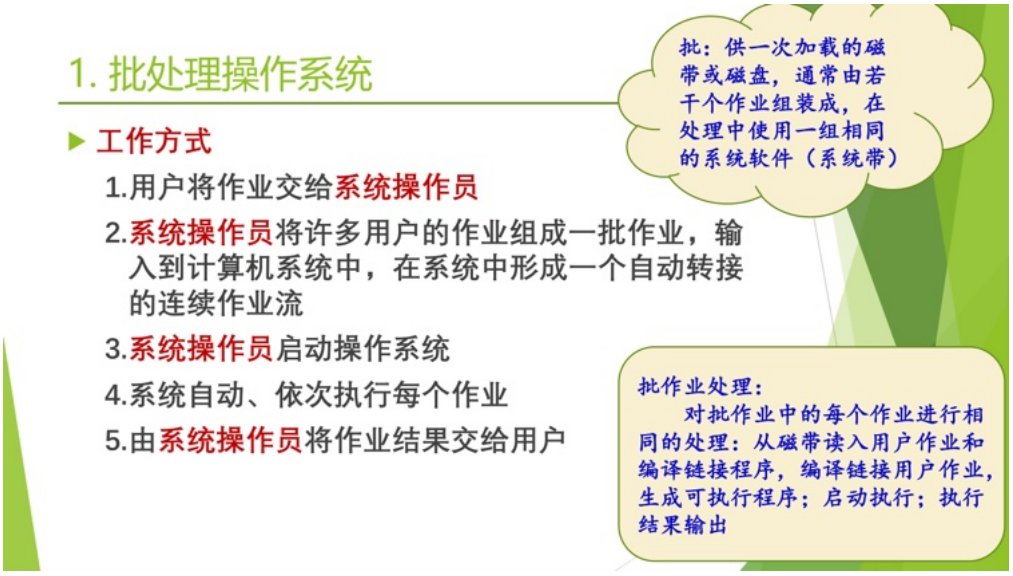
- 关键特征：作业自动顺序执行，无交互。
- 典型作业由作业控制语言、用户程序和数据组成。
- 包括单道批处理与多道批处理两种形态。

- 多道批处理在所有资源维度上都表现出更高的利用率，但需要更复杂的调度和保护机制。

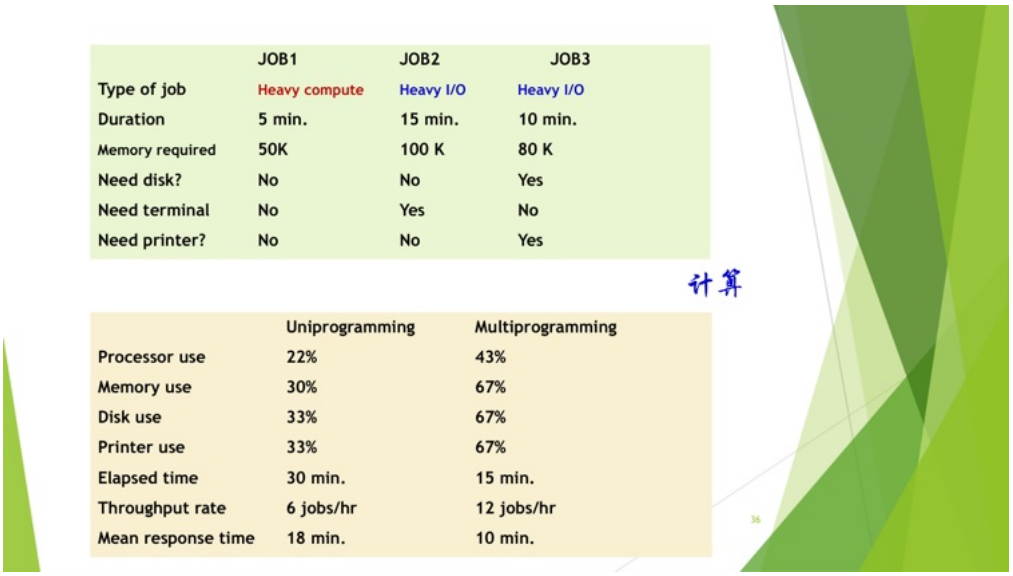
- SPOOLing 把独占设备虚拟成共享设备，提高 I/O 效率，并实现“虚拟设备”这一抽象。

1.7.2 分时操作系统

- **时间片（time slice）**是 CPU 时间被划分出的小片段。



图示：批处理系统的工作流程与作业组织方式。



	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min.	15 min.	10 min.
Memory required	50K	100 K	80 K
Need disk?	No	No	Yes
Need terminal	No	Yes	No
Need printer?	No	No	Yes

计算

	Uniprogramming	Multiprogramming
Processor use	22%	43%
Memory use	30%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min.	15 min.
Throughput rate	6 jobs/hr	12 jobs/hr
Mean response time	18 min.	10 min.

图示：在 CPU、内存、磁盘、打印机等资源上，单道与多道方案的利用率对比。

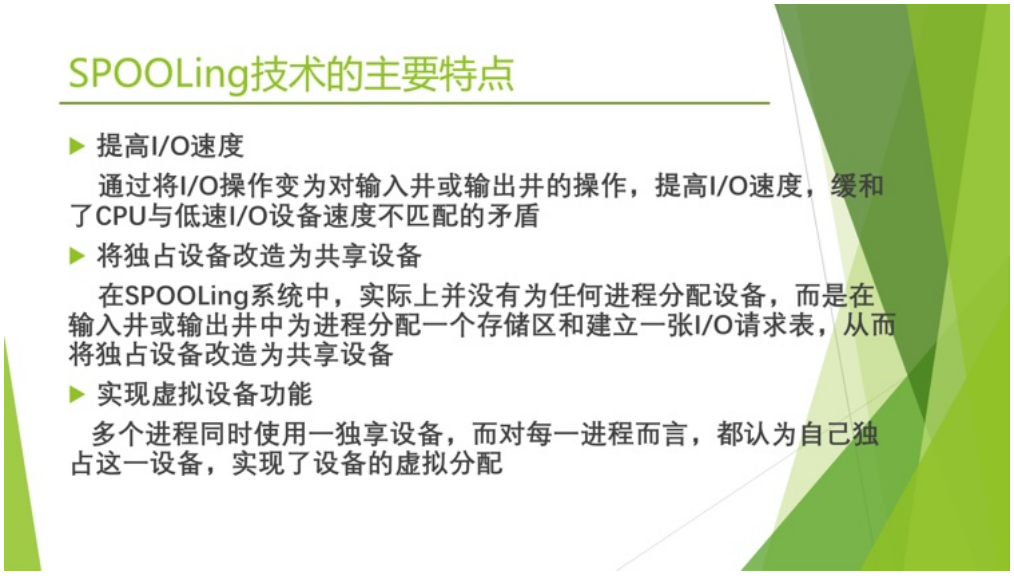
- 操作系统以时间片为单位，轮流为不同终端用户服务。
- 衡量指标是响应时间——从命令发出到系统回应的时间。
- 用户感觉自己独占机器，是“虚拟机”特征在交互场景的具体体现。

1.7.3 实时操作系统

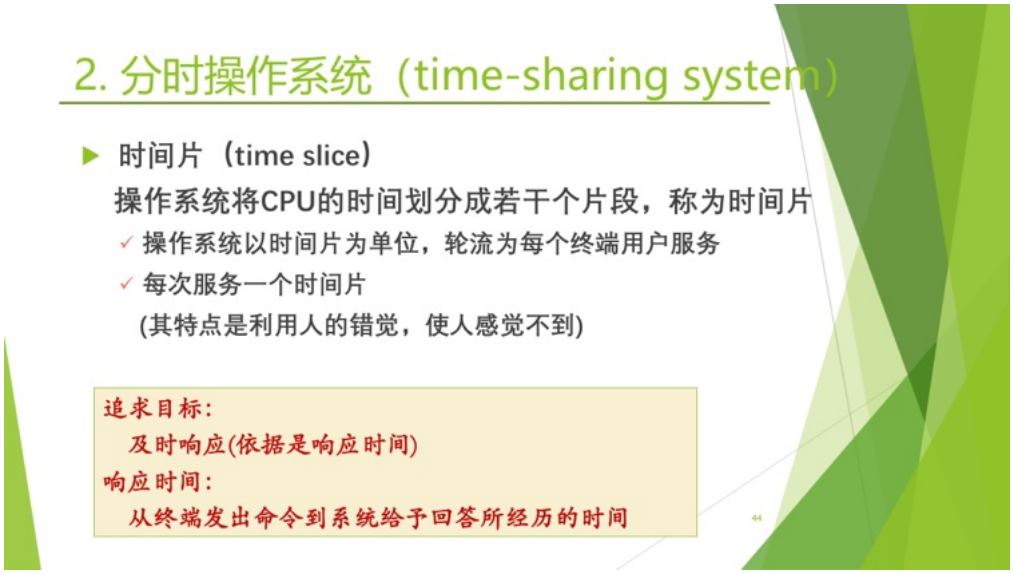
- 关键参数是时间。
- **硬实时**：动作必须严格在规定时间内完成，工业控制、航空电子等。
- **软实时**：偶尔违反时间约束可以接受，例如多媒体播放。

1.7.4 个人计算机操作系统

- 单用户为目标，强调易用性、丰富的应用生态。



图示：SPOOLing 系统通过输入井和输出井缓冲外设。



图示：操作系统把 CPU 时间切成时间片，轮流为终端用户服务。

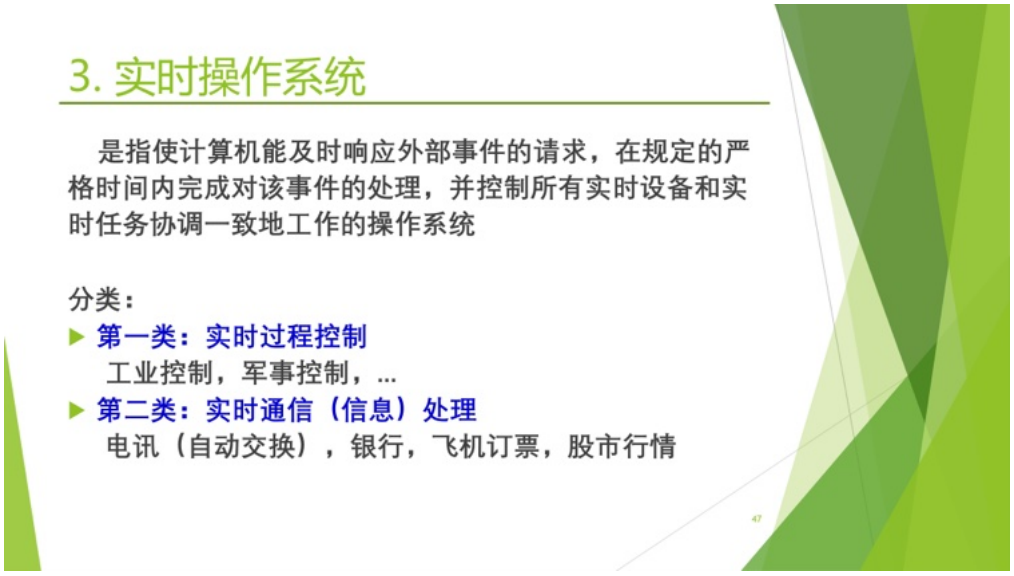
- 典型代表是 Windows、macOS 与桌面 Linux 发行版。

1.7.5 网络操作系统

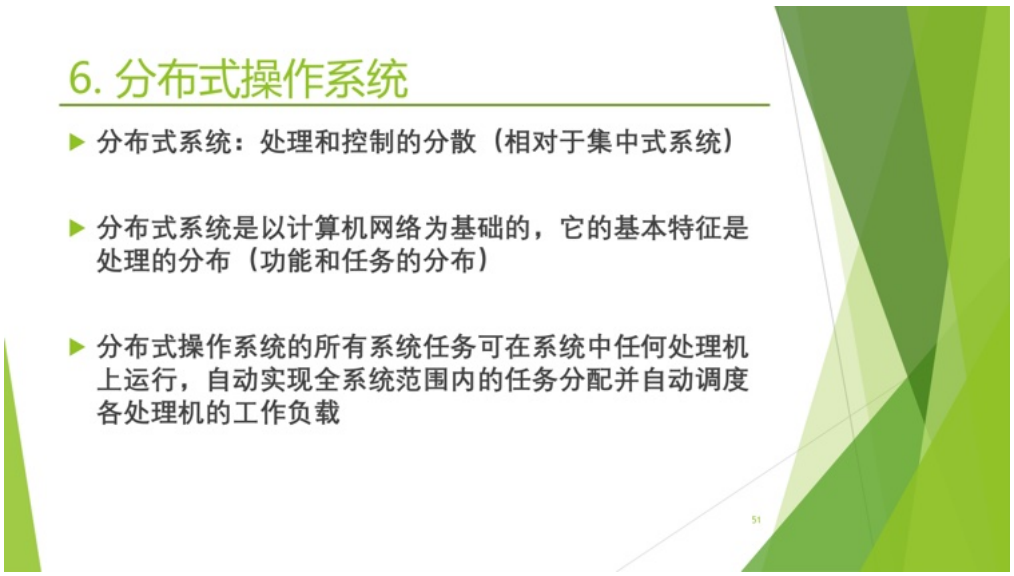
- 在普通操作系统之上叠加网络协议栈、网络管理和安全。
- 目标是相互通信、资源共享。

1.7.6 分布式操作系统

- 多台机器联合形成一个统一的操作系统。
- 关键特征：资源进一步共享、对用户透明、各节点平等自治、整体处理能力增强。



图示：在严格时间约束下做出反应，并保持高可靠性。



图示：处理与控制分散到多台机器上，对用户呈现统一视图。

1.7.7 嵌入式操作系统

- 嵌入到具体设备中，对整个嵌入式系统及其控制对象做协调和调度。
- 通常与实时性强相关，但不总是硬实时。

1.8 重点小结

- 三种视角：资源管理、进程、虚机器。
- 四个特征：并发、共享、虚拟、随机性。
- 重要技术：多道程序设计、中断、通道、SPOOLing、分时、虚拟存储。
- 典型系统：UNIX、MULTICS、OS/360、Windows、Linux、Android、麒麟、鸿蒙、openEuler、统信。



图示：运行在专用设备上，为特定功能服务。



图示：操作系统概念、视角、特征、典型系统与重要技术的全景小结。

- 学完本讲后，应该能在脑中重建一张图：从硬件到应用之间，每一层操作系统机制各自负责什么、彼此如何协作。

LECTURE 2

第 2 讲

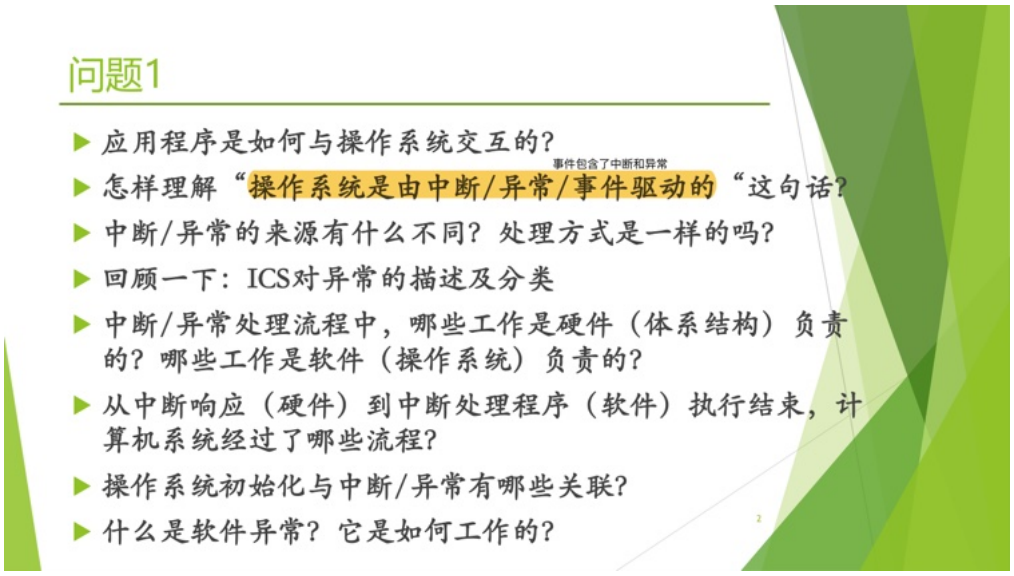
操作系统运行环境和运行机制

Operating Systems Course Notes

2.1 本讲导读

- 这一讲解决两个核心问题：
  1. 操作系统在硬件上**靠什么机制运行**？答案是中断/异常/事件构成的“**异常控制流**（ECF）”。
  2. 用户态程序如何**进入操作系统**？答案是系统调用，而系统调用本质上是受控的异常。
- 这是后面进程管理、调度、内存管理、文件 I/O 等机制的执行基础。理解 ECF 后，再看进程切换、缺页处理、I/O 等都是它的具体应用。
- 本讲对应的“一句话主线”：**用户态进入内核的所有合法路径都汇聚到一个统一的入口，由硬件强制建立、由软件接管处理。**

2.2 开篇问题与核心结论



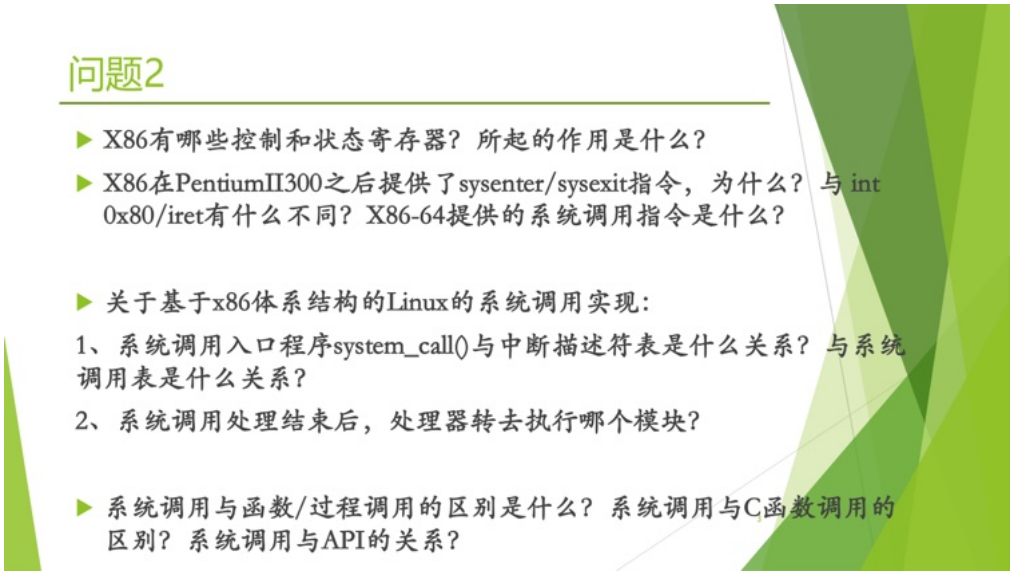
图示：应用程序如何与操作系统交互、为何说 OS 是事件驱动的、中断与异常处理流程的硬件 / 软件分工。

> **核心论断（第 4 页）：**理解 ECF 就理解了应用与操作系统的交互、并发的实现，以及软件异常的工作方式。

2.2.1 八个开篇问题精炼解答

> **思考题 1：**应用程序如何与操作系统交互？

解答：应用程序运行在用户态，不能直接执行特权指令或访问内核数据。所有对操作系统服务的请求都通过 **系统调用（System Call）** 完成，底层由异常机制实现：用户程序执行陷入指令（`int 0x80`、`sysenter`、`syscall`、`ecall`），硬件切换到内核态并跳到统一入口；内核根据系统调用号分派处理函数；完成后用专用返回指令回到用户态。除此之外，无任何合法方式从用户态主动“变成”内核态。



图示：x86 控制 / 状态寄存器、`int 0x80` 与 `sysenter / syscall` 的差别，以及 Linux 系统调用入口的内部组织。



图示：把 ECF 作为本讲的中心概念，连接应用、内核与硬件三方。

> **思考题 2：**为什么说 OS 是中断/异常/事件驱动的?

解答：内核大多数时间不在主动循环运行，而是在最近一次返回用户态后处于等待。一旦发生外设中断、CPU 异常或用户陷入，硬件强制把控制权交给内核相应入口；内核处理完该事件后再返回。系统的执行节奏因此由“事件”决定，而不是由内核自身的循环决定。

> **思考题 3：**中断和异常来源不同，处理方式一样吗?

解答：来源不同：中断是**异步**的，由外设引发；异常是**同步**的，由当前指令本身引发。两者在机制上都通过中断向量分派进入内核，但语义不同：

- 中断处理后通常返回**下一条指令**；

- 故障类异常修复后**重试当前指令**；
- 系统调用（陷入）返回**下一条指令**；
- 终止类异常通常**不返回**，进程被杀死。

> **思考题 4：** ICS 中对异常的分类是什么？

解答： 沿用 Bryant & O'Hallaron《CSAPP》的分类：

- **中断（Interrupt）**：异步、由外设引发；
- **陷入（Trap）**：同步、程序主动安排；
- **故障（Fault）**：同步、可恢复；
- **终止（Abort）**：同步、不可恢复。

> **思考题 5：** 中断/异常处理流程中，硬件和软件分别负责什么？

解答：

- **硬件**：判断事件、保护最少必要现场（PSW、PC、栈指针）、按中断号查中断向量表、必要时切换栈和特权级、进入处理程序入口。
- **软件**：保存其余寄存器、识别具体事件、执行处理逻辑、恢复现场、用专用返回指令回原执行流。

> **思考题 6：** 从中断响应到中断处理结束，系统经历了什么？

解答： 概括为“**响应–保存–分派–处理–恢复–返回**”六步：

1. 设备发出中断或 CPU 检测到异常；
2. 硬件保存最少现场，切换到内核态；
3. 通过中断向量表定位入口；
4. 入口程序保存其它寄存器，调用具体处理函数；
5. 处理结束后恢复现场；
6. 通过 `iret` / `sret` / `mret` 等指令返回原执行流。

> **思考题 7：** 操作系统初始化与中断/异常的关联？

解答： 内核启动必须先建立中断/异常基础设施才能正常运行：

- 初始化中断控制器（PIC / APIC、PLIC 等）；
- 设置中断向量表 / IDT，把每类事件挂到对应入口；
- 注册系统调用入口（设置 IDT 第 0x80 项 / 写 MSR `MSR_LSTAR` / 写 STVEC）；
- 注册时钟中断处理；

- 完成上述工作后，系统进入“中断驱动”的常态运行。

> **思考题 8：**什么是软件异常？

解答：软件异常是程序自身制造或暴露出来的异常，例如除零、空指针解引用、非法指令、`int` 指令、缺页等。硬件检测到这些情况时按异常向量进入内核；操作系统决定如何处理（修复/信号化/终止）。它和外部中断都通过同一套中断分派机制进入内核，差别只在事件来源和语义。

2.2.2 与系统调用相关的开篇问题（第 3 页）

> **思考题：** x86 有哪些控制和状态寄存器？

解答：

- **控制寄存器 CR0–CR4**：CR0 控制保护模式、分页、协处理器等；CR2 在缺页时保存触发地址；CR3 保存页目录基址（PDBR）；CR4 控制更现代的 CPU 特性（PSE、PAE、SMEP、PCID 等）。
- **状态寄存器 EFLAGS**：见 §4.1，存放条件码、IF、IOPL、TF 等。
- **描述符表寄存器**：GDTR、LDTR、IDTR、TR；分别指向 GDT、LDT、IDT 和 TSS。
- **段寄存器**：CS、DS、ES、SS、FS、GS。在保护模式下其内容是“段选择符”，含特权级位 RPL。

> **思考题：**为什么 PentiumII 300 之后引入 `sysenter / sysexit`？它和 `int 0x80 / iret` 有何不同？x86-64 用什么？

解答：

- 早期使用 `int 0x80` 进入内核存在性能开销：要查 IDT、做段选择符与 GDT 的特权级检查、压入完整的中断栈帧。
- `sysenter / sysexit` 通过专用 MSR 直接保存内核入口、内核栈指针，省去对 IDT 与段描述符的查询，是“快速系统调用”。语义上要求调用方提前布置返回点，因此约定比 `int 0x80` 更严格。
- 区别：`int 0x80` 走完整异常路径、`sysenter` 走快速路径；`sysenter / sysexit` 不需要保存恢复 EFLAGS 中的中断标志，整体上下文切换开销更小。
- x86-64 上使用 `syscall / sysret`，本质思想与 `sysenter` 相同，但配套的 MSR 不同（`MSR_LSTAR`、`MSR_STAR`、`MSR_SFMASK`）。

> **思考题：** Linux 系统调用入口程序 `system_call()` 与 IDT、`sys_call_table` 的关系？

解答：

- IDT 第 128（0x80）项是一个**陷阱门**，其入口偏移指向 `system_call()` 的第一条指令。
- 用户执行 `int $0x80` 时，硬件按 IDT 跳到 `system_call()`。

- `system_call()` 保存现场后，把 `eax`（系统调用号）作为索引，从 `sys_call_table` 中读取对应内核函数地址并调用。
- 因此 IDT 把“事件”分派到“入口”，`sys_call_table` 把“系统调用号”分派到“处理函数”。两个表分工明确。

> **思考题：**系统调用处理结束后处理器去哪?

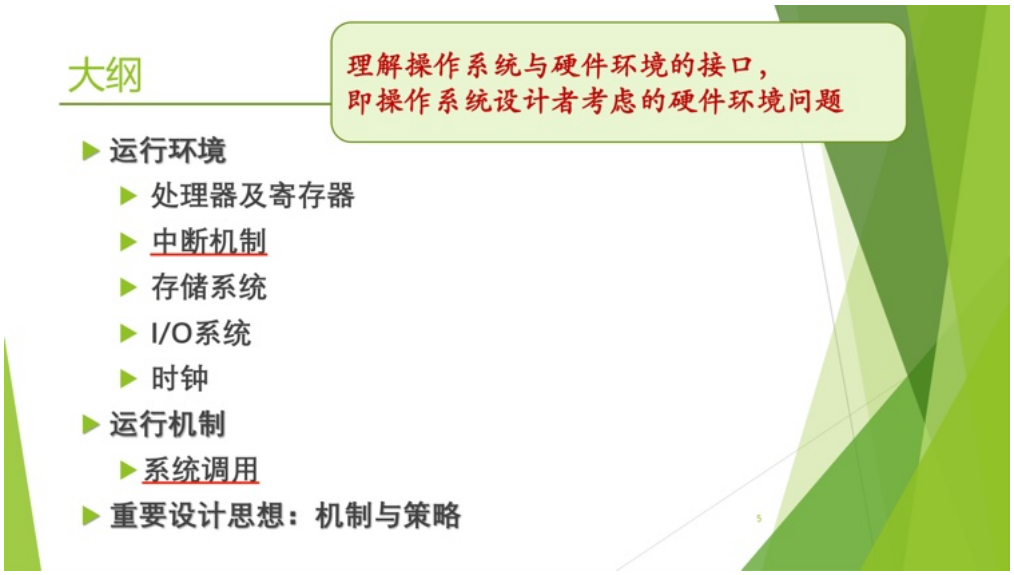
解答：通过 `ret_from_sys_call`：检查是否需要重新调度、是否有未处理信号；然后 `RESTORE_ALL` 把保存的寄存器恢复，并执行 `iret`（或 `sysret`）返回用户态原指令之后。

> **思考题：**系统调用与函数/过程调用、C 库、API 的关系?

解答：

- **C 函数调用**：纯用户态的栈操作和跳转，开销小，不能访问内核资源。
- **API**：面向应用的接口规范（如 POSIX）。
- **C 库函数**：实现 API 的用户态封装。许多 API 在 C 库中实际只是封装了系统调用（如 `write`、`open`、`fork`），也有完全用户态的（如 `strlen`、`memcpy`）。
- **系统调用**：API 中需要内核服务的部分，必须切换特权级。

2.3 本讲大纲

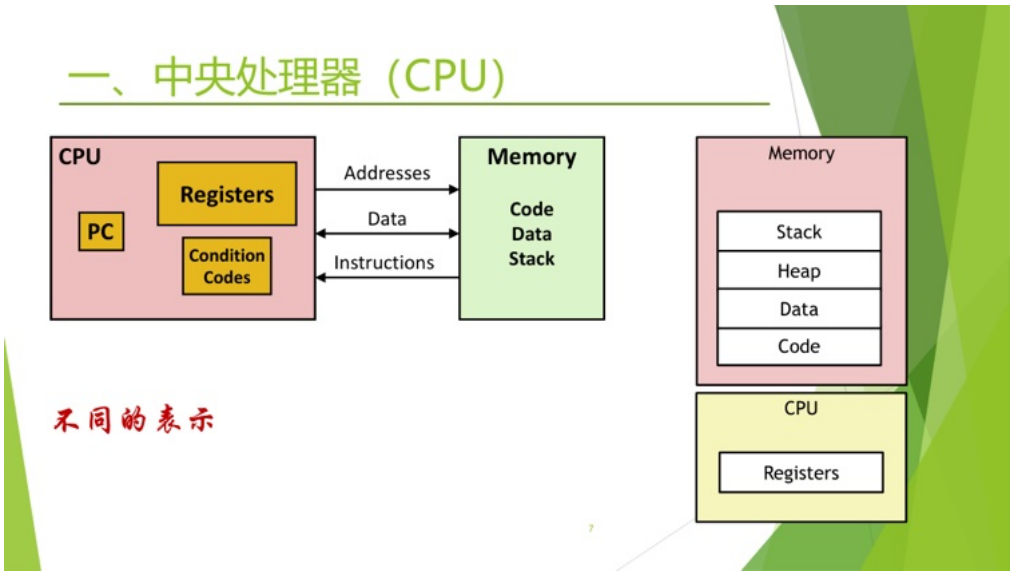


图示：左侧是“运行环境”——处理器、寄存器、中断、存储、I/O、时钟；右侧是“运行机制”——系统调用、机制与策略分离。

要把握的不是知识点列表，而是从硬件机制到软件抽象的因果链：

- 硬件提供：特权级、寄存器、中断/异常、地址保护、时钟。
- 操作系统在这些机制之上构建：进程、内存、I/O、调度。

2.4 中央处理器与寄存器



图示：CPU 内含 PC、通用寄存器、控制寄存器、缓存等部件，对外通过地址 / 数据 / 指令总线访问内存。

2.4.1 寄存器分类



图示：用户可见寄存器（数据 / 地址 / 条件码）与控制和状态寄存器（PC、IR、PSW）。

- **用户可见寄存器**：机器语言可直接访问。
  - **数据寄存器（通用寄存器）**：算术与逻辑运算的工作区。
  - **地址寄存器**：保存指针，与寻址相关。
  - **条件码寄存器**：保存最近一次运算的零、进位、溢出等标志位。
- **控制和状态寄存器**：用于控制处理器操作，多在特权级下访问。
  - **程序计数器 PC**：下一条要取的指令地址。
  - **指令寄存器 IR**：当前正在执行的指令。

  - **程序状态字 PSW**：见 §3.2。
  - **段/页表基址寄存器、TSS 选择符（TR）、IDTR、GDTR**：管理保护模式下的内存与中断结构。
- 处理器在物理上由“运算器 + 控制器 + 寄存器组 + 高速缓存”构成。

2.4.2 PSW 的作用

PSW 是“一字概括 CPU 状态”的寄存器，集中保存：

- **条件码标志**：零标志（Zero）、进位（Carry）、溢出（Overflow）、符号（Sign）等。
- **中断控制位**：IF（中断使能）、IOPL（I/O 特权级）等，控制是否响应外部中断。
- **处理器模式位**：当前 CPU 特权级，区分用户态 / 内核态。
- **内存管理相关标志**：分页是否启用、保护模式是否启用等。

PSW 在中断/异常发生时由硬件保存，进程切换时由内核保存到 PCB；返回时再恢复，从而保证执行流的“位置 + 模式”一致。这是 OS 实现特权保护和进程隔离的核心硬件机制之一。

2.4.3 x86 控制寄存器 CR0–CR4

寄存器	关键作用
CR0	PE 位启用保护模式；PG 位启用分页；其他位控制协处理器等
CR1	Intel 保留
CR2	缺页时保存触发该次缺页的线性地址
CR3	页目录基址寄存器（PDBR），切换进程时随页表一起切换
CR4	现代特性开关：PSE、PAE、PCID、SMEP、SMAP 等

这些寄存器都是“特权级 0 才能写”的资源，操作系统通过它们配置硬件行为。

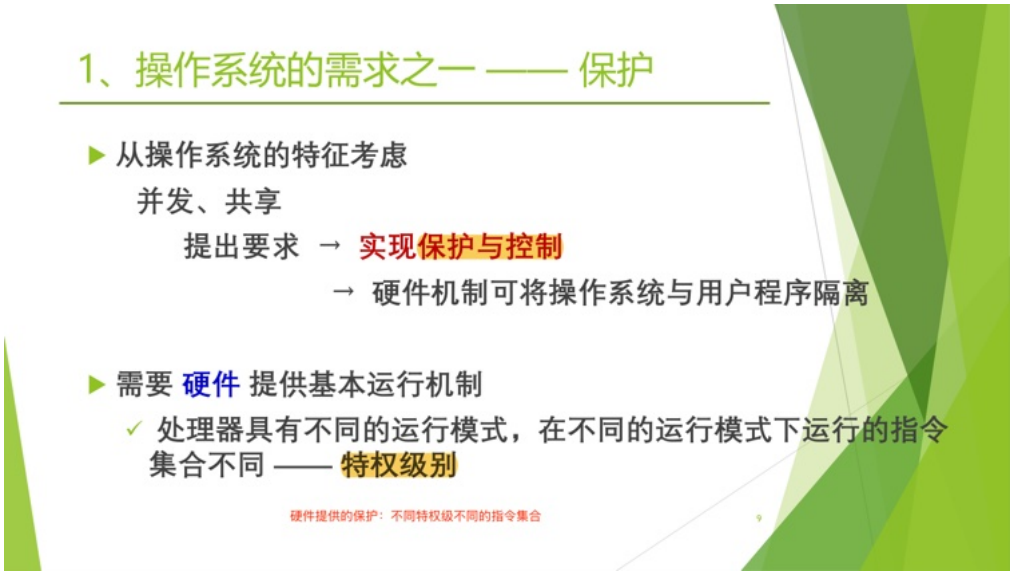
2.5 保护机制：特权级与状态切换

2.5.1 特权级的本质与 EFLAGS

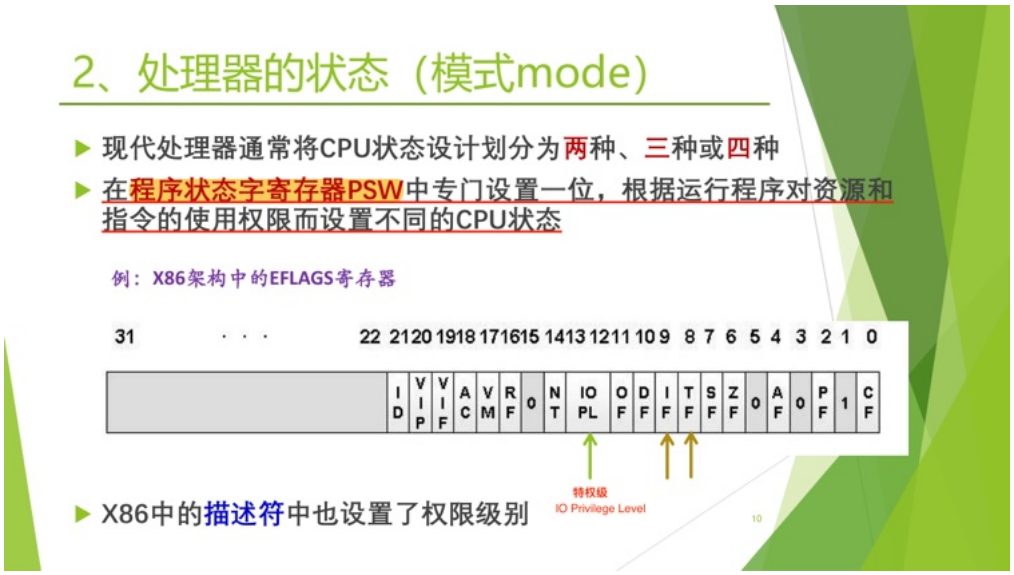
- 特权级是“硬件读取 PSW，决定该指令是否被允许”的具体化。
- 不同特权级允许不同的指令集合：高特权级可访问全部寄存器、所有内存、I/O；低特权级只能执行非特权指令、访问受限内存。

2.5.2 特权指令与非特权指令

- **特权指令**：仅内核态可用，例如修改页表寄存器、关中断、I/O 指令、写控制寄存器等。
- **非特权指令**：用户态可用，例如算术、逻辑、访存（受地址保护约束）。
- x86 设计了 4 个特权环 R0–R3，但主流操作系统只使用 R0（内核）和 R3（用户）。



图示：从并发与共享出发，得出“硬件必须提供保护机制隔离 OS 与用户程序”的结论。

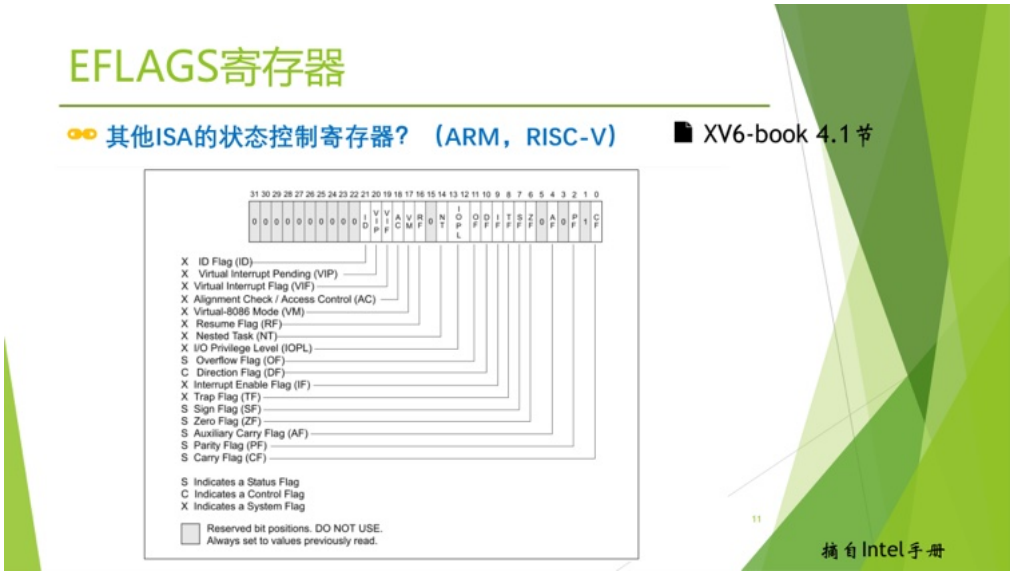


图示：处理器在 PSW（如 EFLAGS）中通过特定位指示当前模式，硬件按位决定可执行的指令集合。

2.5.3 用户态与内核态的切换

- **用户态 -> 内核态**：唯一合法路径是中断、异常、陷入。任何越权尝试都会被硬件转化成保护异常。
- **内核态 -> 用户态**：操作系统执行专用返回指令（`iret` / `sysret` / `sret` / `mret`），同时把 PSW 中的特权级位设回用户态。
- 常见陷入指令：`int`、`trap`、`syscall`、`sysenter` / `sysexit`、`ecall` 等。

> **要点：**用户态不能“自己变成内核态”。任何越权尝试都会触发异常，由内核接管，因此特权切换是被硬件强制约束的。



图示：EFLAGS 中 IF（中断使能）、IOPL（I/O 特权级）、TF（陷阱位）等位决定保护行为。



图示：内核态运行操作系统代码可使用全部指令；用户态只能使用非特权指令。

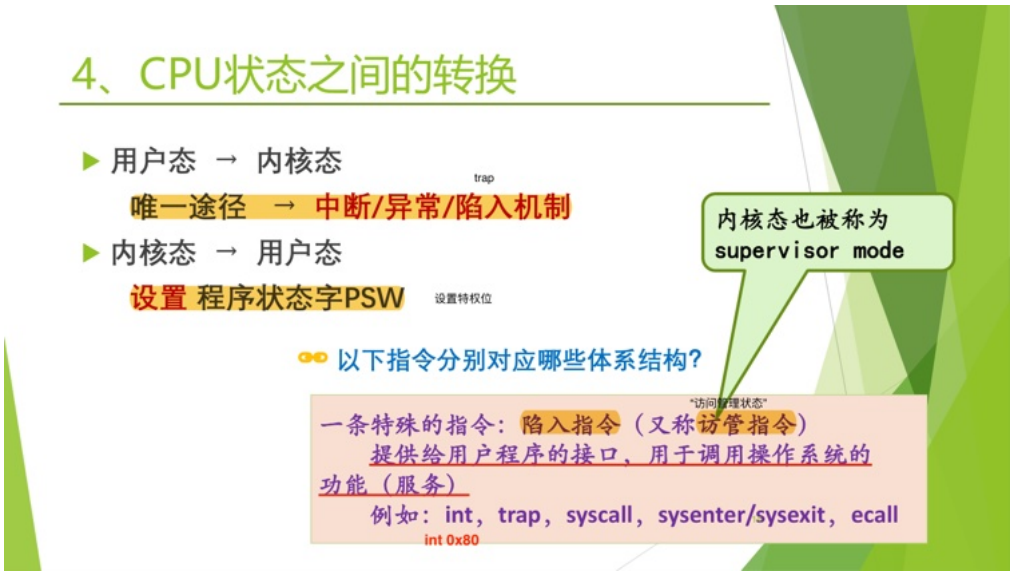
2.6 中断机制与事件分类

2.6.1 中断/异常的概念与特点

- 中断/异常是 CPU 对系统中“某个事件”做出的反应。
- 三大特点：
  - **随机发生**：发生时机不可由用户程序控制；
  - **自动处理**：硬件主动完成现场保存与跳转；
  - **可恢复**：处理结束后能回到原执行流。
- 一句话：**事件改变了处理器的控制流**。



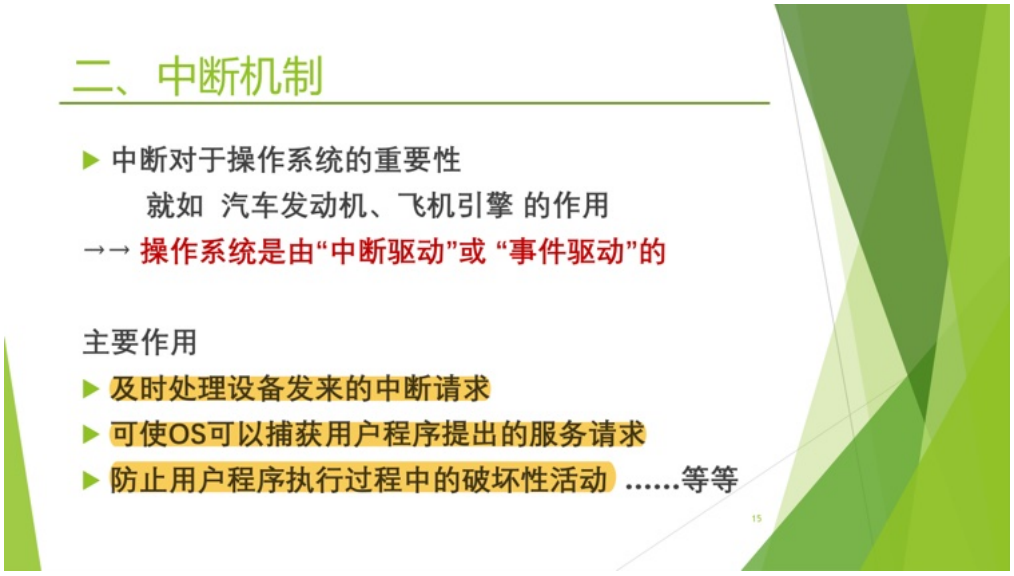
图示：M（机器态）、S（监管态）、U（用户态）。XV6-riscv 在 S 模式下运行内核，在 U 模式下运行用户程序。



图示：用户态进入内核态只能通过中断/异常/陷入；返回时由内核显式设置 PSW。

2.6.2 引入原因（历史）

- **中断的引入**：早期 I/O 与 CPU 串行造成大量等待。引入中断后，CPU 启动设备工作即可去做别的事；设备完成时通过中断通知 CPU。
- **异常的引入**：用统一机制处理算术错误、保护错误、缺页、陷入指令等指令运行时问题，避免在每条指令后做大量错误检查。



图示：中断之于操作系统，相当于发动机之于汽车——OS 是“中断驱动”或“事件驱动”的。



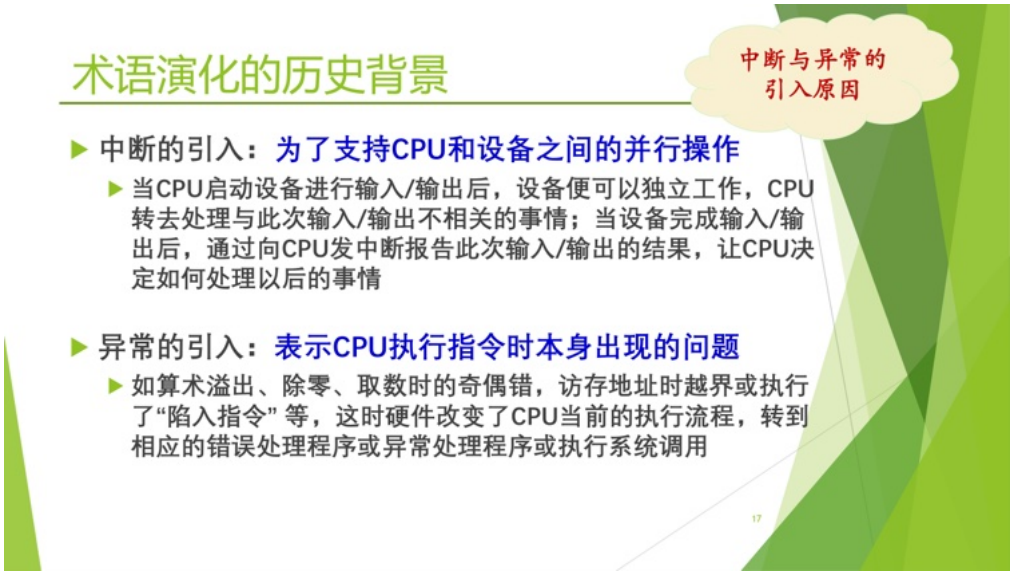
图示：CPU 暂停当前程序、保存现场、跳到事件处理程序、处理后恢复现场，整个过程对程序透明。

2.6.3 中断与异常的来源（细化）

类别	典型来源
中断（外中断）	键盘、鼠标、网络接口卡、定时器、硬盘控制器、其他 I/O 设备
异常（内中断）	除零、非法指令、保护错误、缺页、断点指令、调试事件、`int` 指令、其他程序性错误
系统调用	实质是陷入异常的一种

处理方式上：

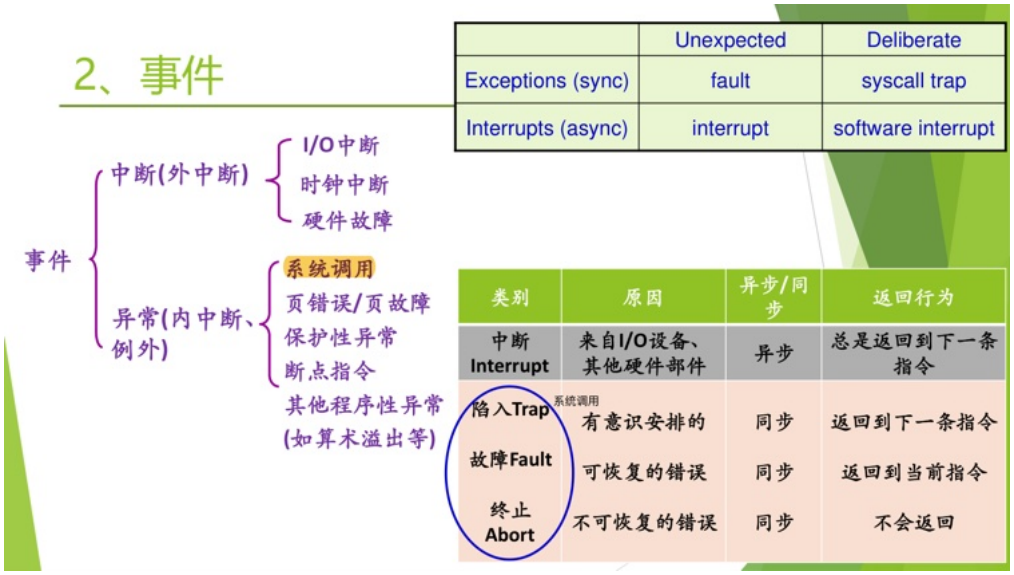
- 中断处理：响应外设请求，保存现场 -> 处理（读寄存器、清中断、操作设备）-> 恢复现场 -> 返回。
- 异常处理：识别异常类型 -> 修复（如缺页换入）或终止 -> 决定返回当前指令、下一条指令



图示：中断为了支持 CPU 与设备并行工作；异常为了反映指令执行本身的问题。

或不返回。

2.6.4 事件分类总表

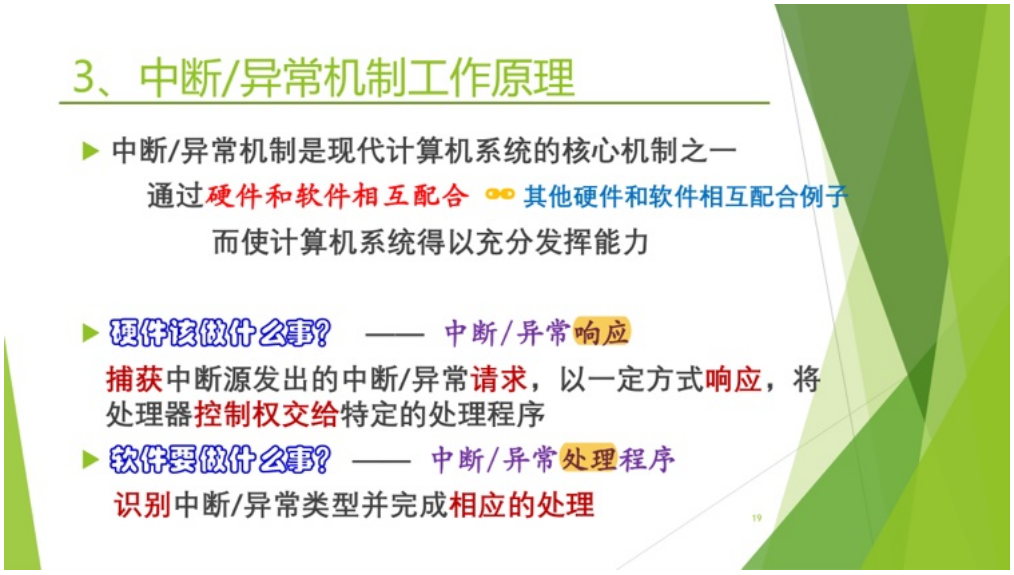


图示：中断、陷入、故障、终止四类事件的来源、同步性和返回行为。

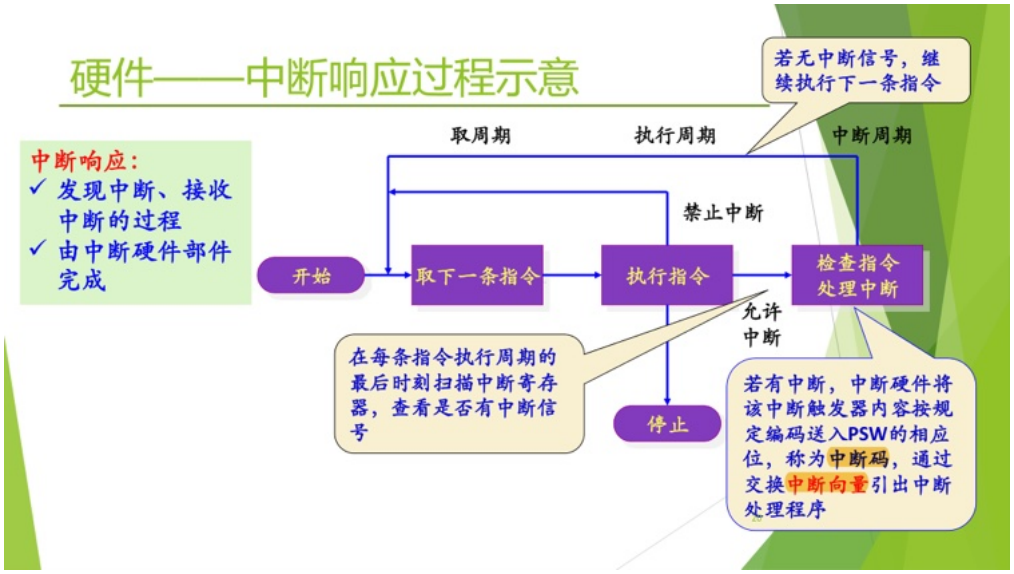
类型	来源	同步性	返回行为
中断 Interrupt	来自 I/O 设备、其他硬件部件	异步	返回到下一条指令
陷入 Trap	有意识安排的（系统调用、断点）	同步	返回到下一条指令
故障 Fault	可恢复的错误（如缺页）	同步	返回到当前指令
终止 Abort	不可恢复的错误(如硬件错误)	同步	不会返回

2.7 中断/异常机制的工作原理

2.7.1 硬件视角：响应过程



图示：中断/异常机制是硬件与软件配合的产物——硬件做响应，软件做处理。

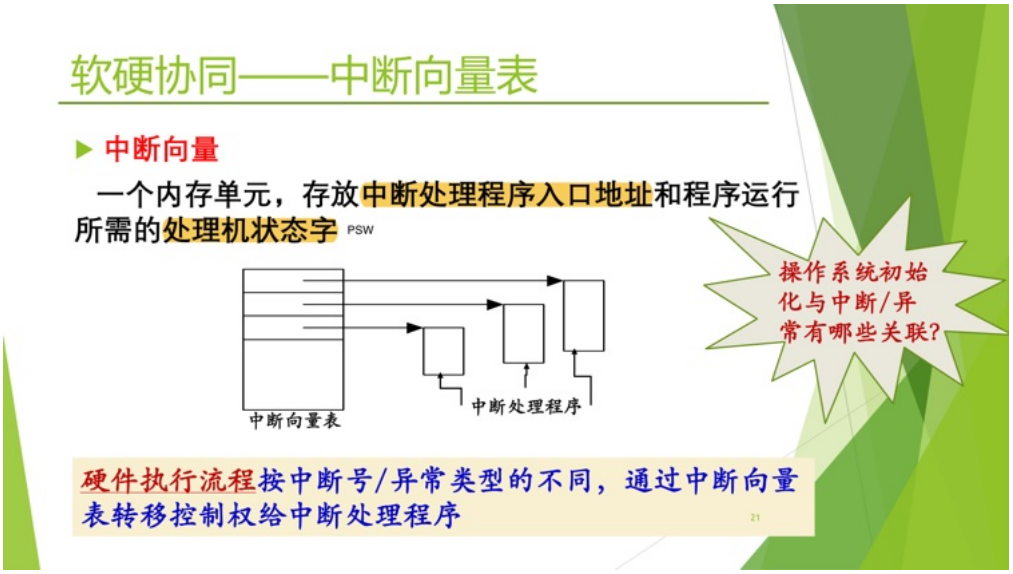


图示：每条指令执行的最后阶段扫描中断；若有中断信号且中断使能位允许，硬件进入中断周期。

具体动作：

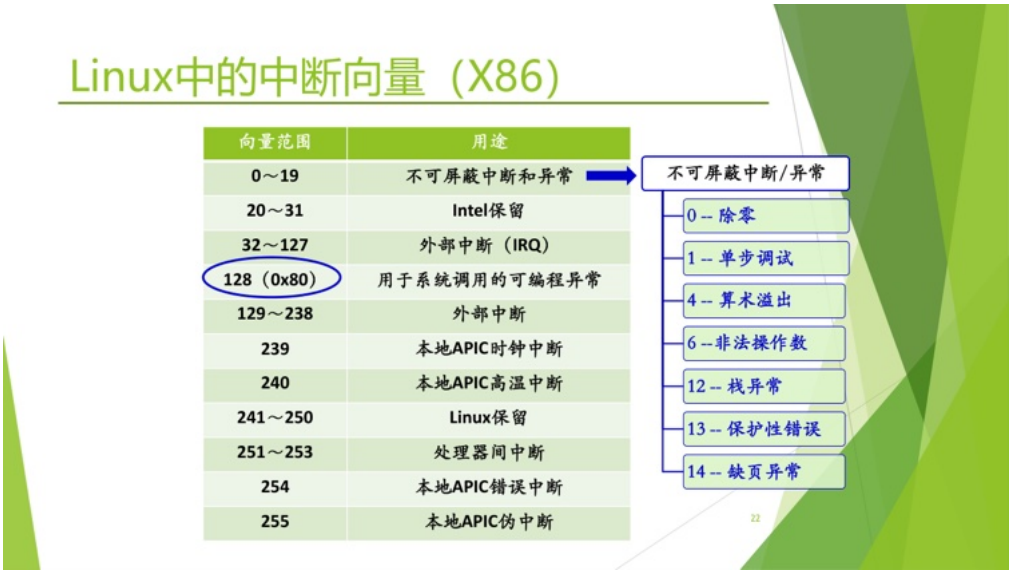
1. 在每条指令执行的最后阶段，CPU 扫描中断寄存器；
2. 若中断被禁止或无中断信号，进入下一条指令；
3. 若有中断信号且允许：
   - 把中断触发器内容编码送入 PSW 的相应位（中断码）；
   - 通过中断向量表交换出中断处理程序入口；
   - 切换到入口处开始执行。

2.7.2 软硬协同：中断向量表



图示：硬件按中断号在中断向量表中找到入口地址 + 处理机状态字，跳到对应处理程序。

- **中断向量** = 入口地址 + PSW（必要的处理机状态）。
- **中断向量表 / IDT** 是按中断号索引的数组。
- 操作系统初始化阶段必须把每一类事件挂到这个表上，否则硬件无法分派。



图示：Linux 在 x86 上对 0~255 号中断的具体分配。

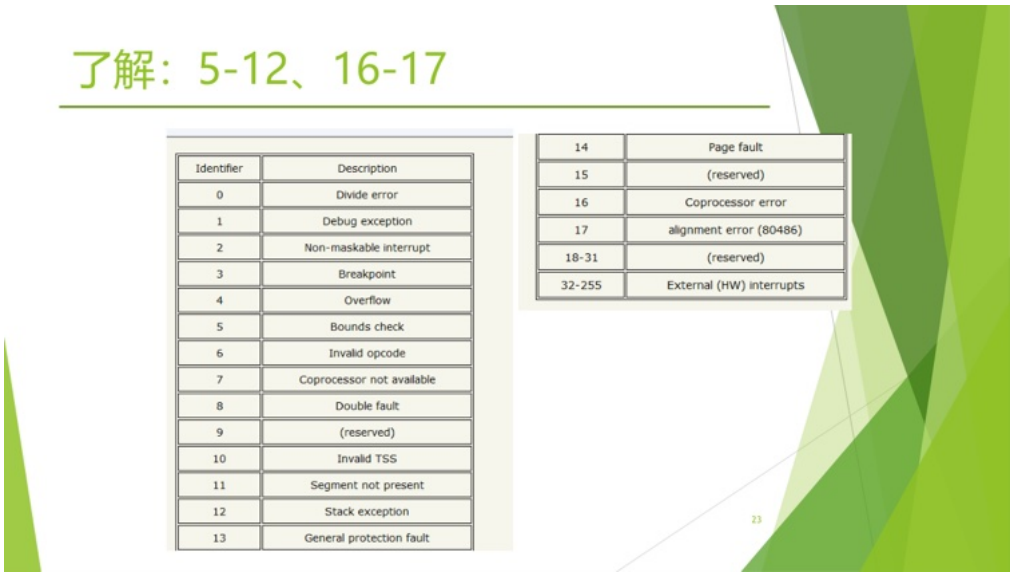
不可屏蔽中断与异常（前 20 号）记忆要点：

- 0：除零（#DE）；1：调试（#DB）；2：NMI；3：断点（#BP）；4：溢出（#OF）；5：边界检查（#BR）；
- 6：无效操作码（#UD）；7：设备不可用（#NM）；8：双重故障（#DF）；9：协处理器段越界；
- 10：无效 TSS（#TS）；11：段不存在（#NP）；12：栈段错误（#SS）；

- 13：通用保护（#GP）；**14：缺页（#PF），是后续虚存章节的核心异常**；
- 15：保留；16：浮点（#MF）；17：对齐检查（#AC）；18：机器检查（#MC）；19：SIMD 浮点（#XF）。

其余区段：

- 20–31：Intel 保留；
- 32–127：外部中断 IRQ；
- **128（0x80）：Linux 系统调用陷阱门**；
- 239：本地 APIC 时钟中断；240：高温；251–253：处理器间 IPI；254/255：APIC 错误 / 伪中断。



图示：考试和复习重点关注 5–12、16–17 与 14（缺页）。

2.7.3 中断响应的 5 步

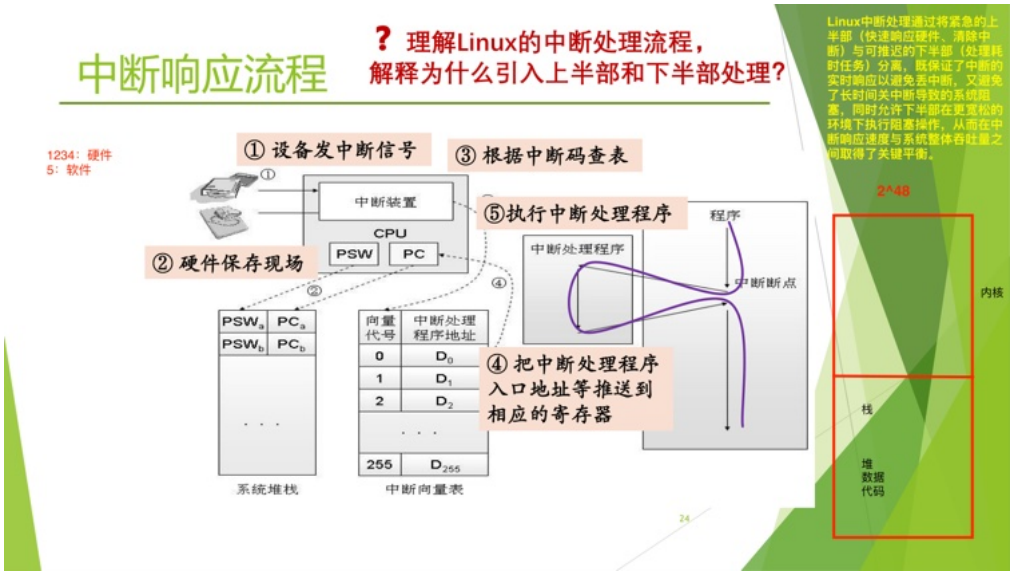
> **要点：**第 1 ~ 4 步由硬件完成，第 5 步起由软件接管。

2.7.4 上半部与下半部

> **思考题：**为什么 Linux 把中断处理拆成上半部和下半部?

解答：

- 中断处理程序运行在中断上下文，处理时常常关中断，**执行时间过长会丢失其他中断**。
- 上半部（Top Half）：**硬中断**部分，紧急、必须立刻完成，例如读寄存器、清中断、记录关键信息、调度下半部。它运行时通常关中断，不能阻塞，也不能进行复杂操作。
- 下半部（Bottom Half）：**软中断 / tasklet / 工作队列**，可推迟，可被进一步打断，可在更宽松的环境下执行复杂处理（例如解析数据包、唤醒等待进程）。



图示：(1) 设备发中断信号 (2) 硬件保存现场 (3) 根据中断码查表 (4) 把入口地址等推入寄存器 (5) 执行中断处理程序。

- 在 Linux 中，每个 CPU 都有一个软中断内核线程 `ksoftirqd/<cpuid>`，专门负责执行软中断。

设计目的：

1. **响应快**：上半部尽量短，让系统能尽快响应下一个中断；
2. **关中断时间短**：减少系统并发瓶颈；
3. **紧急/非紧急分离**：把紧急任务放上半部、耗时任务放下半部，提高整体吞吐量。

例子：网卡收包时，硬件产生中断 -> 上半部禁网卡中断、读取最少必要信息、触发软中断 -> 下半部以 ksoftirqd 解析协议、唤醒应用。

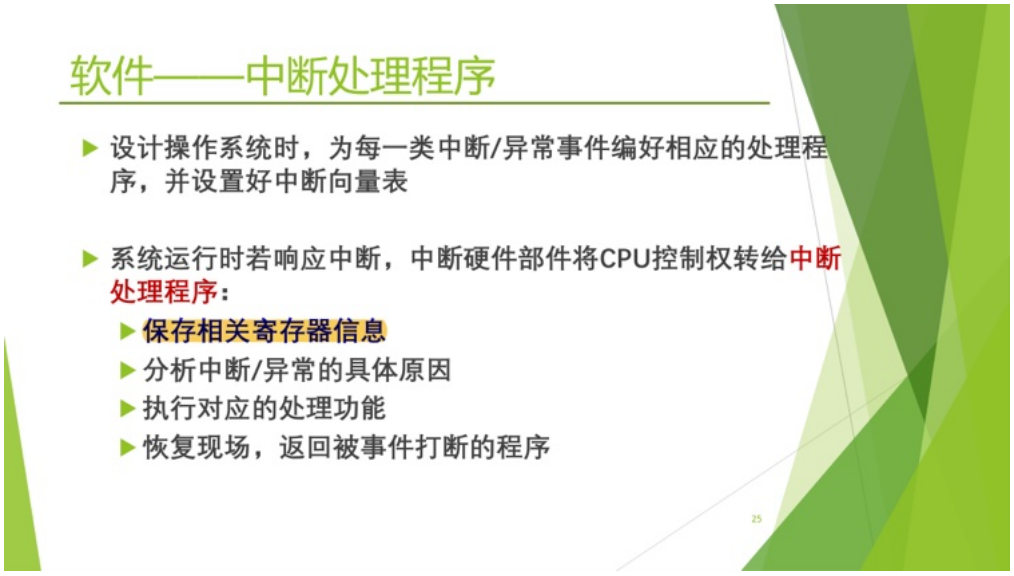
2.7.5 软件视角：中断处理程序

操作系统在初始化时为每一类事件准备处理程序，并把它们写入中断向量表。运行时按下面四步处理：

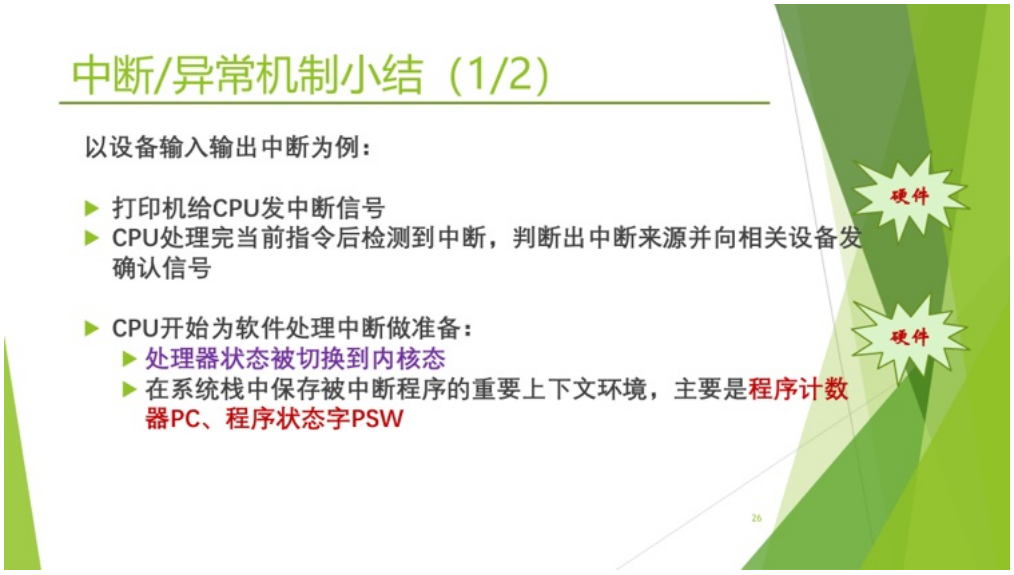
1. 保存相关寄存器信息；
2. 分析中断/异常的具体原因；
3. 执行对应的处理逻辑；
4. 恢复现场，返回被打断的程序。

2.7.6 完整流程小结（以打印机中断为例）

> **结论：**中断/异常机制不是“硬件单干”或“软件单干”，而是“硬件做最少必要步骤，软件做剩下所有事”的协作。



图示：保存现场 -> 识别原因 -> 执行处理 -> 恢复现场。



图示：打印机发中断 -> CPU 处理完当前指令 -> 切换内核态 -> 系统栈保存 PC、PSW。

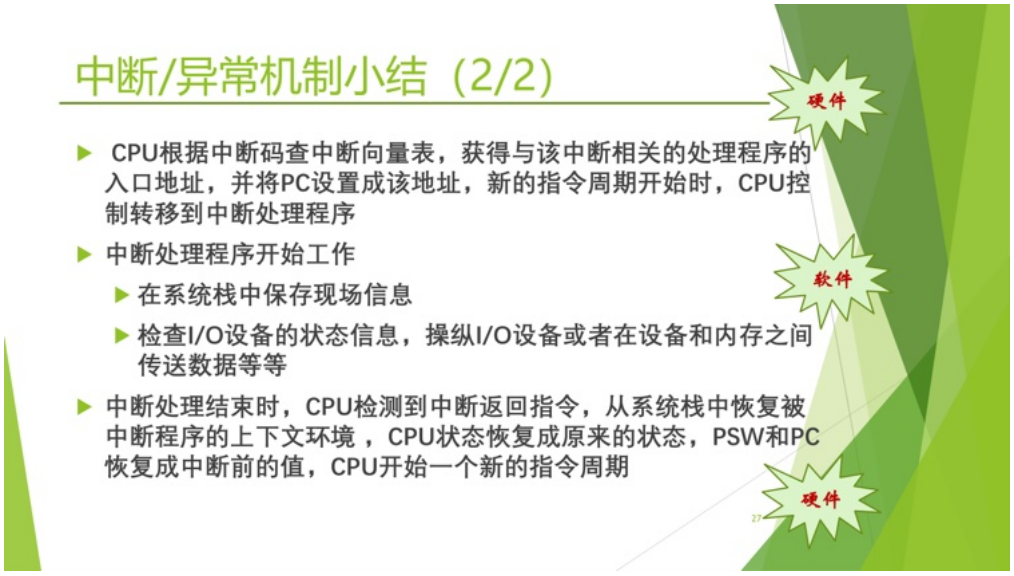
2.8 IA-32 体系结构对中断的支持

2.8.1 基本概念

- **中断**：由硬件信号引发，分为可屏蔽（INTR）与不可屏蔽（NMI）。
- **异常**：由指令执行引发。某些异常会在执行处理程序前压入硬件出错码到内核态堆栈。
- **系统调用**：异常的一种，是用户态进入内核态的唯一入口。

2.8.2 实模式 vs 保护模式

- **中断控制器（PIC / APIC）**：把硬件中断信号转换为中断向量，引发 CPU 中断。
- **实模式中断向量表**：表项是“段地址左移 4 位 + 偏移”，**不支持 CPU 状态切换**，与一般过程调用相似。



图示：查中断向量表 -> 跳到处理程序 -> 保存现场、操作设备 -> 中断返回指令恢复并返回原程序。



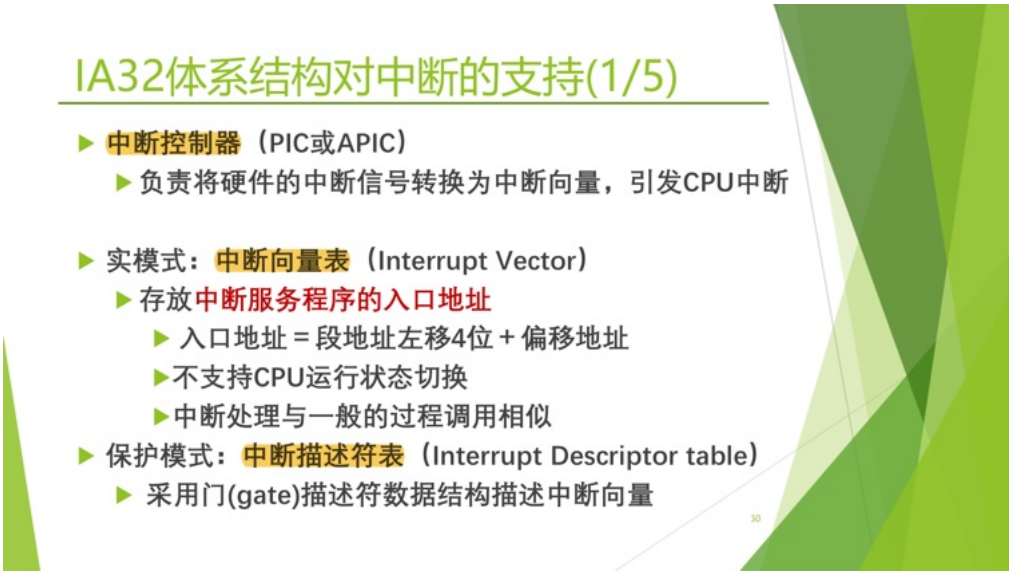
图示：x86 区分中断（硬件信号、可屏蔽 / 不可屏蔽）、异常（指令执行引发，~20 种）和系统调用（异常的特例）。

- **保护模式 IDT**：表项是“门描述符”，专门描述如何从用户态切到内核态。

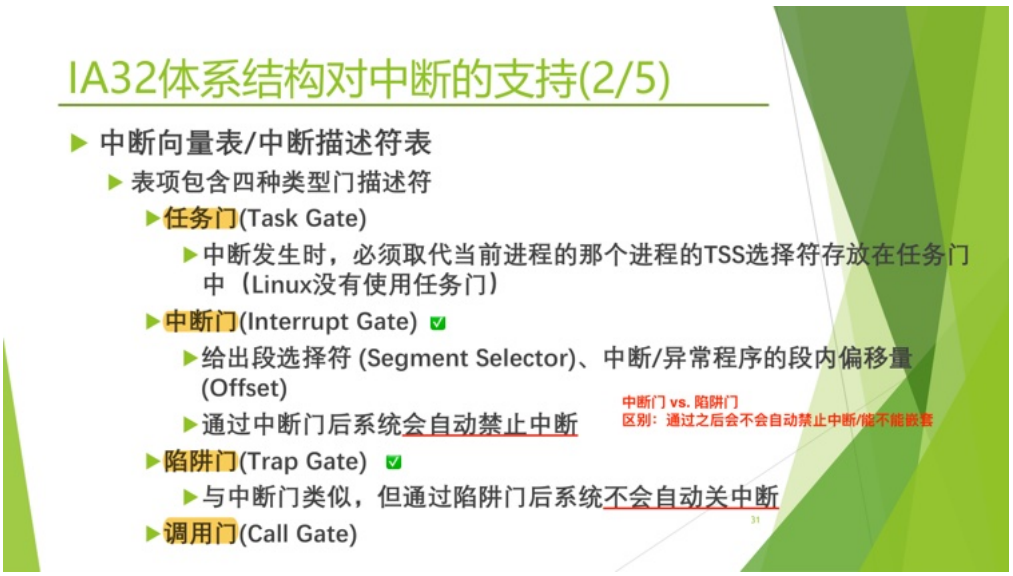
2.8.3 IDT 中的四种门描述符

- **任务门（Task Gate）**：进入时切换到由 TSS 描述的另一个任务。Linux 没有使用它。
- **中断门（Interrupt Gate）**：跳入处理程序后**自动禁止中断**，避免嵌套。
- **陷阱门（Trap Gate）**：与中断门类似，但**不会自动关中断**，允许嵌套，便于调试与陷入。
- **调用门（Call Gate）**：在不同特权级之间显式调用某段代码，使用较少。

> **关键比较：**中断门 vs. 陷阱门只差在“是否自动关中断”这一点上，因此 Linux 用陷阱门挂系统调用入口（系统调用执行时不必关中断）。



图示：中断控制器 PIC / APIC、实模式中断向量表、保护模式 IDT 与门描述符。



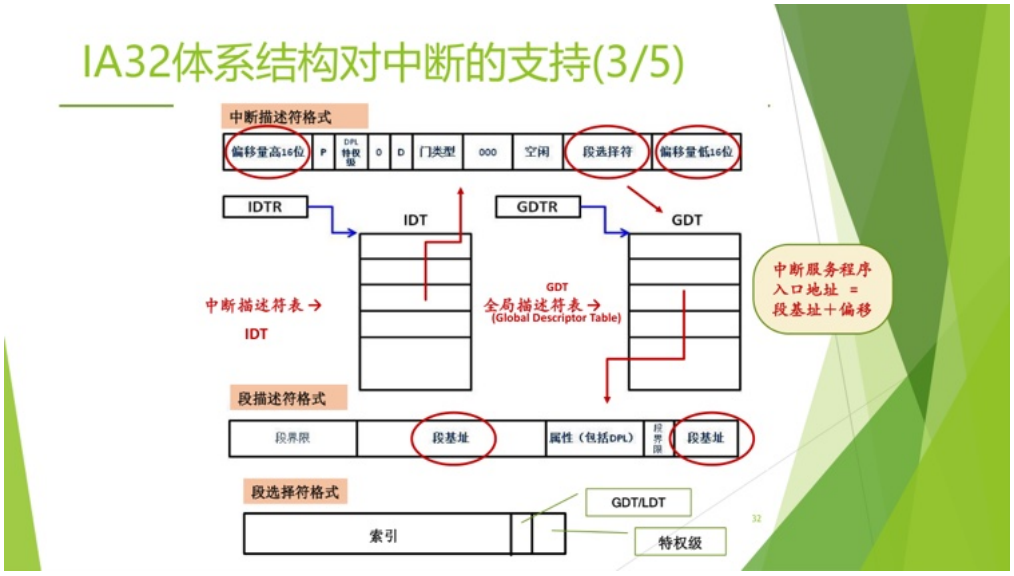
图示：任务门、中断门、陷阱门、调用门。

2.8.4 段选择符与 IDT/GDT 联动

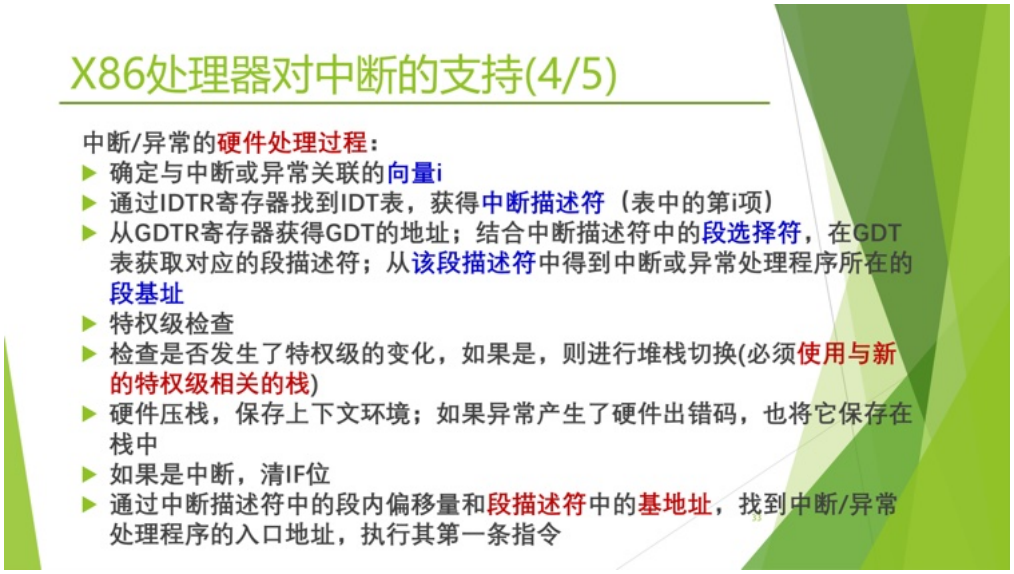
- 段选择符由 13 位索引、TI 位（GDT/LDT）、2 位 RPL（请求特权级）组成。
- IDT 中的中断描述符给出段选择符 + 段内偏移；
- GDT 中通过该段选择符的索引取出段描述符，得到段基址；
- 中断处理程序的入口 = 段基址 + 偏移。

2.8.5 IA-32 的硬件处理过程（8 步）

1. 确定与中断或异常关联的向量 i。
2. 通过 IDTR 找到 IDT，取出第 i 项中断描述符。
3. 由 GDTR 找到 GDT，结合中断描述符中的段选择符取出段描述符，得到处理程序所在段的



图示：段选择符 = 索引 + GDT/LDT 标志 + 特权级 RPL；IDT 中存中断描述符，GDT 中存段描述符。

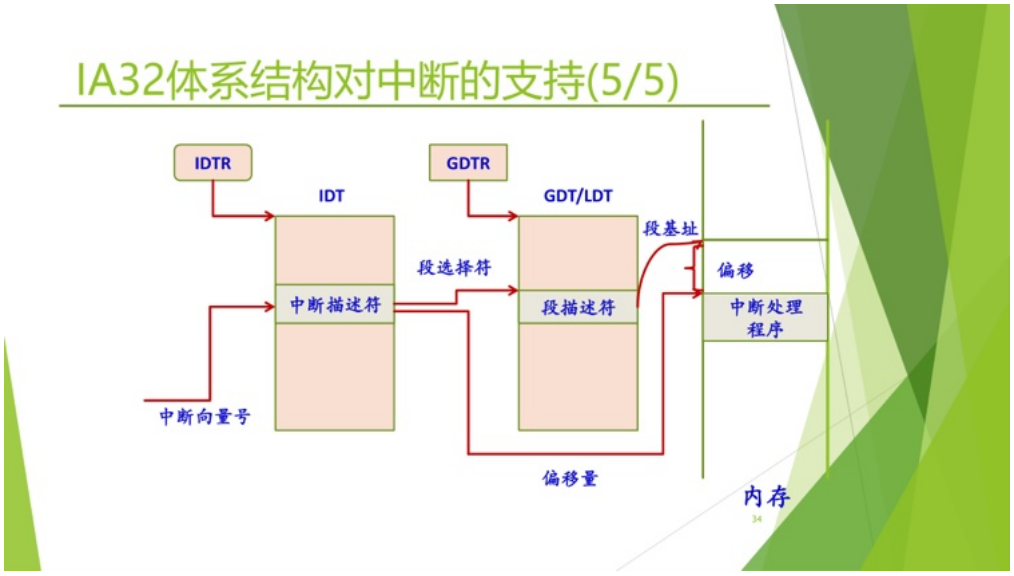


图示：硬件处理过程的具体顺序。

基址。

4. 进行特权级检查。
5. 若发生特权级变化，从 TSS 中取新栈指针，进行**栈切换**。
6. 硬件压栈保存上下文（用户 SS:ESP、EFLAGS、用户 CS:EIP、出错码等）。
7. 若是中断（不是陷阱），自动 **清 IF 位**关中断。
8. 通过段基址 + 段内偏移定位中断/异常处理程序的第一条指令并执行。

2.9 系统调用机制



图示：IDTR -> IDT -> 中断描述符 -> GDT -> 段描述符 -> 中断处理程序。



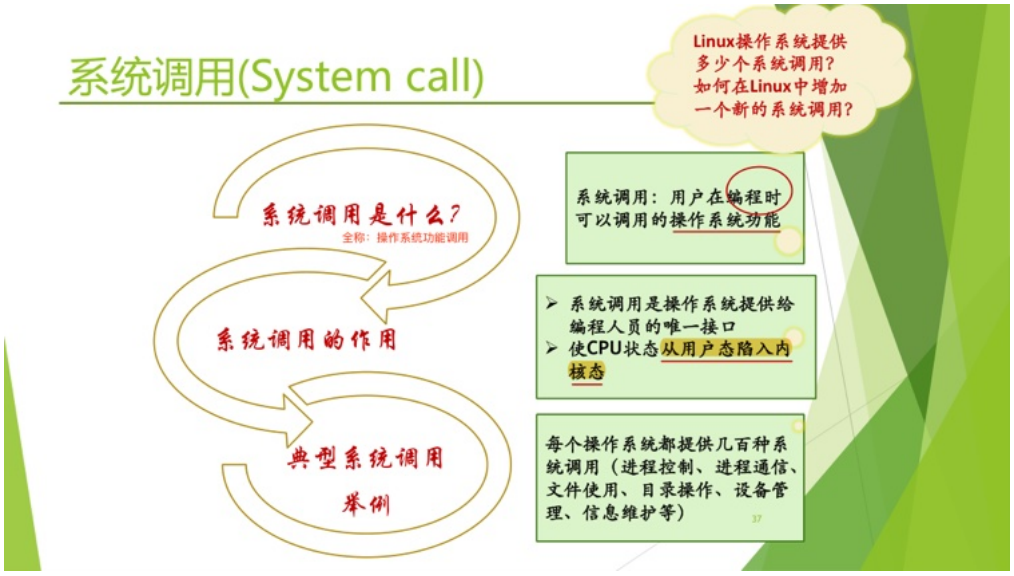
图示：本讲第二条主线开始——系统调用的作用。

2.9.1 系统调用是什么

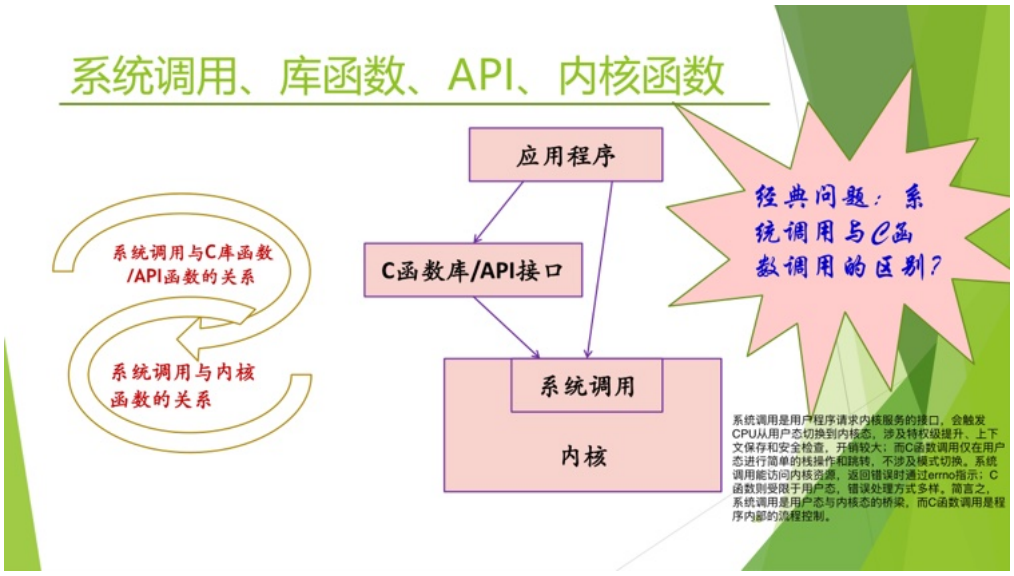
- **系统调用**又称“操作系统功能调用”。
- 是用户程序进入内核的**唯一合法入口**。
- 调用时 CPU 状态由用户态切换为内核态。
- 操作系统通过几百种系统调用提供进程控制、进程通信、文件使用、目录操作、设备管理、信息维护等。

2.9.2 系统调用 / 库函数 / API / 内核函数

- **C 库函数**通常是封装层，可能直接执行系统调用，也可能完全在用户态完成（例如 `strlen`）。
- **API** 是面向应用的接口规范（如 POSIX）。



图示：系统调用是用户在编程时可以使用的操作系统功能，是 OS 给编程人员的唯一接口，会让 CPU 状态从用户态陷入内核态。



图示：应用调用 C 库函数 -> 库函数封装系统调用 -> 内核函数完成实际工作。

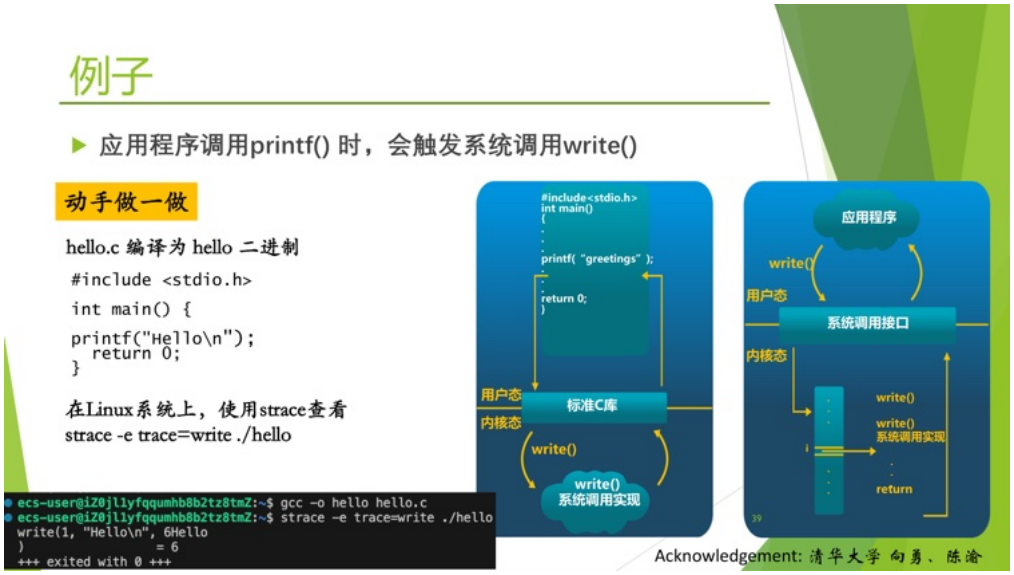
- **系统调用**是 API 中需要内核服务的部分。
- **内核函数**是系统调用最终执行的内核内部实现。

> **核心区别：**系统调用要切换特权级、保存上下文、走异常控制流；普通 C 函数调用只在用户态做栈操作和跳转。前者开销大、能访问内核资源；后者开销小，但不能直接访问内核。`printf` 是 C 库函数，本身在用户态；它最终调用 `write` 系统调用进入内核完成实际输出。

2.9.3 系统调用机制的设计

实现一个系统调用机制需要四块基础设施：

1. **陷入指令**：用一条特殊指令引发异常完成切换，例如 `int 0x80`、`syscall`、`sysenter`、`ecall`。



图示：用户调用 `printf` -> 库内部调用 `write` 系统调用 -> 内核 `write` 实现。`strace -e trace=write ./hello` 可观察调用过程。



图示：本节将分为“静态机制设计”和“动态执行流程”两段。

2. **系统调用号**：所有系统调用走同一入口，通过编号区分要执行的功能。
3. **参数传递**：寄存器、陷入指令自带、内存堆栈。
4. **系统调用表**：以系统调用号为索引的函数指针表，存放每个内核处理函数的入口。

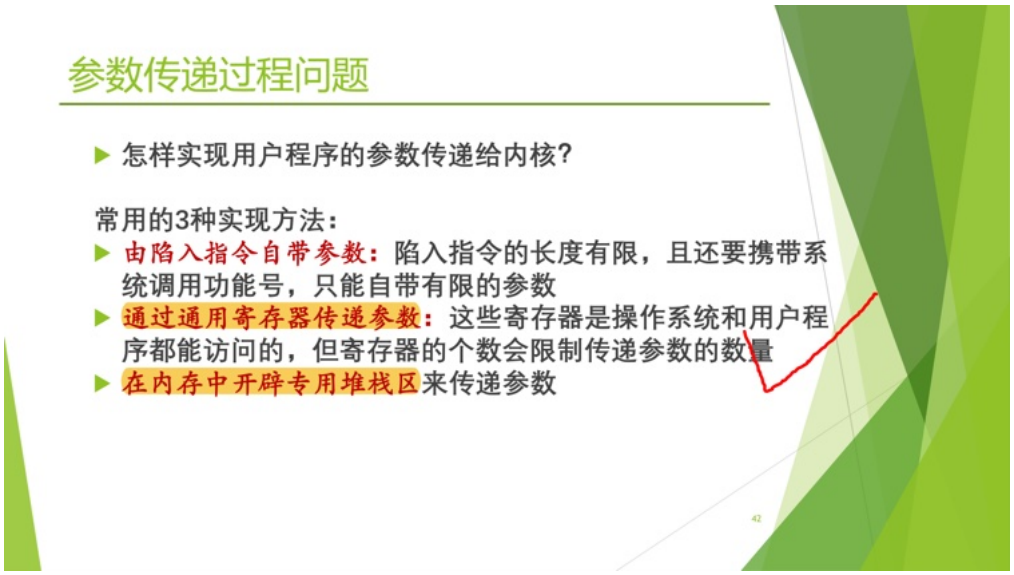
> **重点：**系统调用进来的“口”是同一个，中断向量号相同；但功能不同——这就是“机制与策略分离”：进入入口是**机制**，根据系统调用号选具体处理函数是**策略**。

2.9.4 参数传递的三种方式

- **陷入指令自带参数**：长度有限，仅能传少量参数。
- **通用寄存器传参**：x86 32 位 Linux 使用 `eax`（系统调用号 + 返回值），`ebx`，`ecx`，`edx`，`esi`，`edi`（参数 1–5）。最多约 6 个参数。



图示：陷入指令、系统调用号 + 参数、系统调用表，加上“机制与策略分离”原则。



图示：陷入指令自带、通用寄存器传递、内存中专用堆栈区。

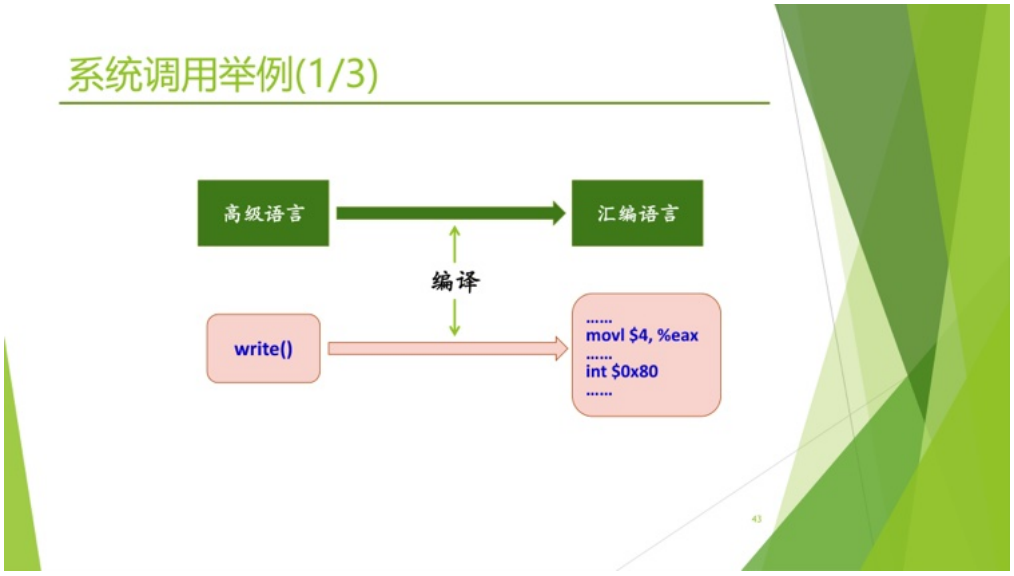
- **内存堆栈区**：参数过多时由用户程序填好后把指针传给内核。

2.9.5 系统调用的高级语言/汇编视角

C 程序：

```c
#include <unistd.h>
int main(){
    char string[7] = {'H','e','l','l','o','!','\n'};
    write(1, string, 7);
    return 0;
}
```

汇编（数据段）：



图示：高级语言中的 write() 编译后翻译为 movl $4, %eax + int $0x80。

```
section .data
output:
    .ascii "Hello!\n"
output_end:
    .equ len, output_end - output
```

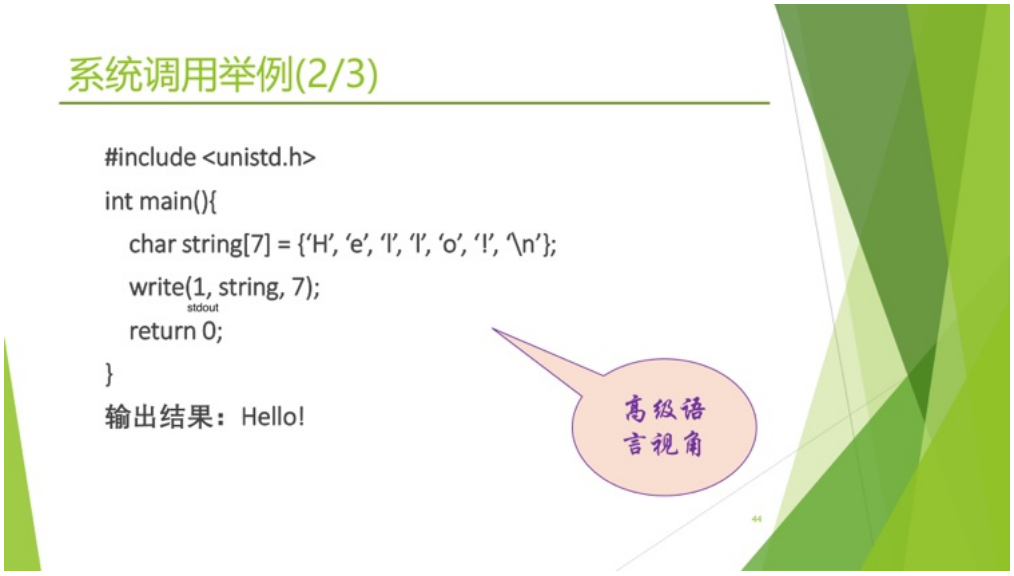
- output：标签为字符串起始。
- .equ len, output_end - output 用差值定义字符串长度，等同于 7。

汇编（代码段）：

```
section .text
.globl _start
_start:
    movl $4, %eax       # 4 是 write 的系统调用号
    movl $1, %ebx       # 文件描述符 1 (stdout)
    movl $output, %ecx  # 字符串地址作为缓冲区
    movl $len, %edx     # 字符串长度
    int  $0x80          # 引发系统调用 write
end:
    movl $1, %eax       # 1 是 exit 的系统调用号
    movl $0, %ebx       # 退出码 0
    int  $0x80          # 引发系统调用 exit
```

执行结果：

```
Hello!
```



图示：调用 `write(1, string, 7)`。



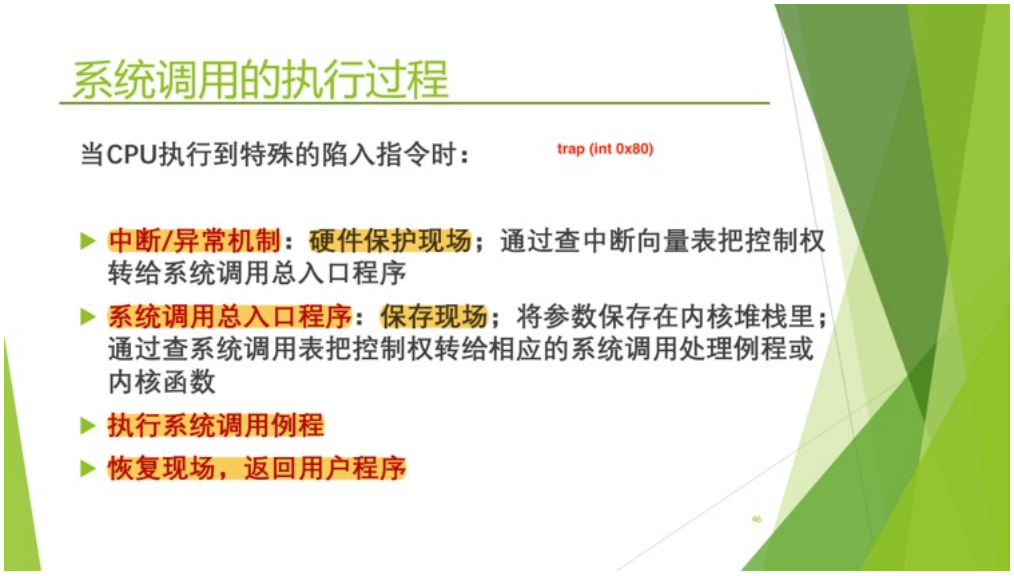
图示：用 `int $0x80` 引发系统调用，传参与系统调用号都通过寄存器准备。

2.9.6 系统调用的执行过程

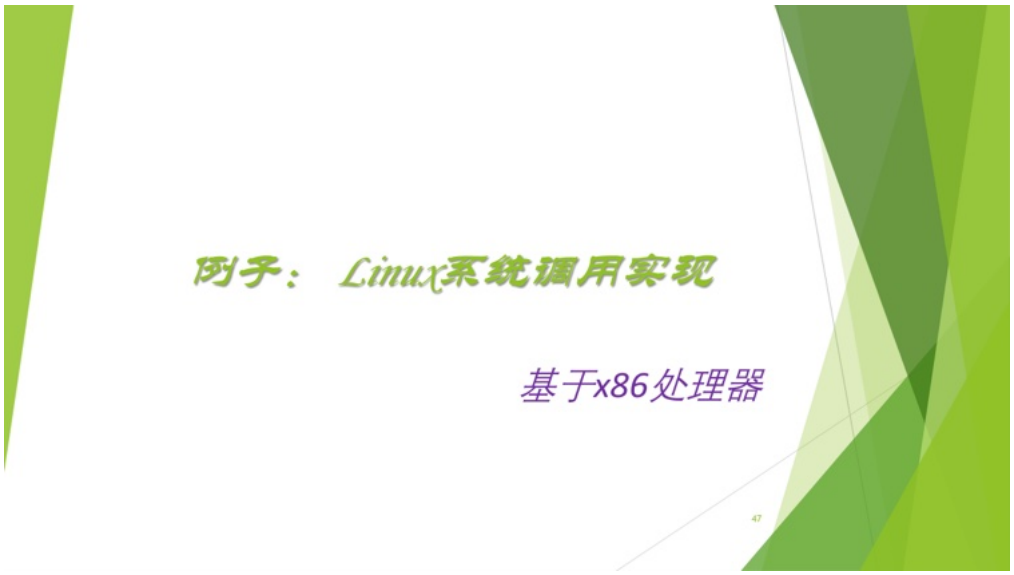
可以拆成五步：

1. 用户程序执行陷入指令；
2. 中断/异常机制：硬件保存现场，通过 IDT 把控制权交给系统调用总入口；
3. 总入口程序：保存现场，把参数放入内核栈，根据系统调用号查表分派；
4. 调用对应内核函数完成具体工作；
5. 恢复现场，使用专用返回指令回到用户态。

2.10 Linux 系统调用实现（x86 视角）



图示：陷入指令 -> 总入口 -> 查系统调用表 -> 调用具体例程 -> 返回。



图示：基于 x86 的 Linux 系统调用实现示例。

2.10.1 入口与门描述符

- 内核初始化阶段调用 `set_system_gate(0x80, &system_call)`，把 IDT 的第 128 项设为陷阱门：
  - 段选择符为内核代码段 `__KERNEL_CS`；
  - 偏移指向 `system_call()` 第一条指令；
  - **DPL = 3**：允许用户态访问该门描述符；
  - **门类型 = 15**：陷阱门，**进入后不会自动关中断**，允许在系统调用执行时仍响应其他中断；
- 因此用户态执行 `int $0x80` 能合法进入内核，并落到 `system_call` 入口。

2.10.2 系统调用号示例



图示：内核初始化时调用 `set_system_gate(0x80, &system_call)`，把 IDT 的第 128 项配置为陷阱门。

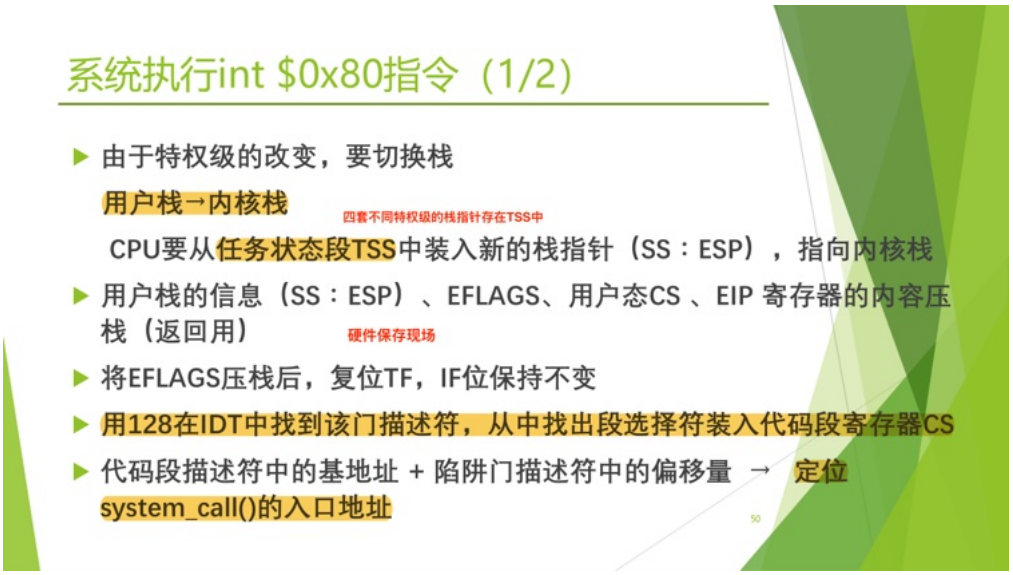


图示：`include/asm-i386/unistd.h` 中的系统调用号宏定义。

```c
#define __NR_exit    1
#define __NR_fork    2
#define __NR_read    3
#define __NR_write   4
#define __NR_open    5
#define __NR_close   6
#define __NR_waitpid 7
#define __NR_creat   8
#define __NR_link    9
#define __NR_unlink  10
#define __NR_execve  11
#define __NR_chdir   12
#define __NR_time    13
/* ... */
```

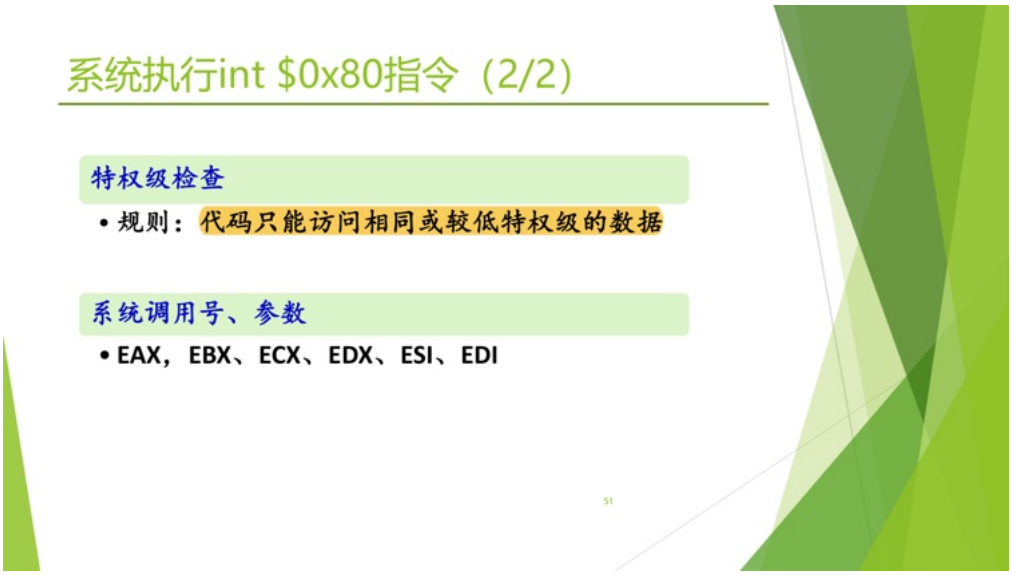
- 这些宏给所有系统调用编号，是 `sys_call_table` 的索引。
- 在用户态，C 库通常按照这些编号填 `eax` 后再 `int $0x80`。

2.10.3 `int $0x80` 的硬件动作（详细两步）



图示：硬件保存现场——栈切换、压入 SS:ESP、EFLAGS、CS:EIP，复位 TF。

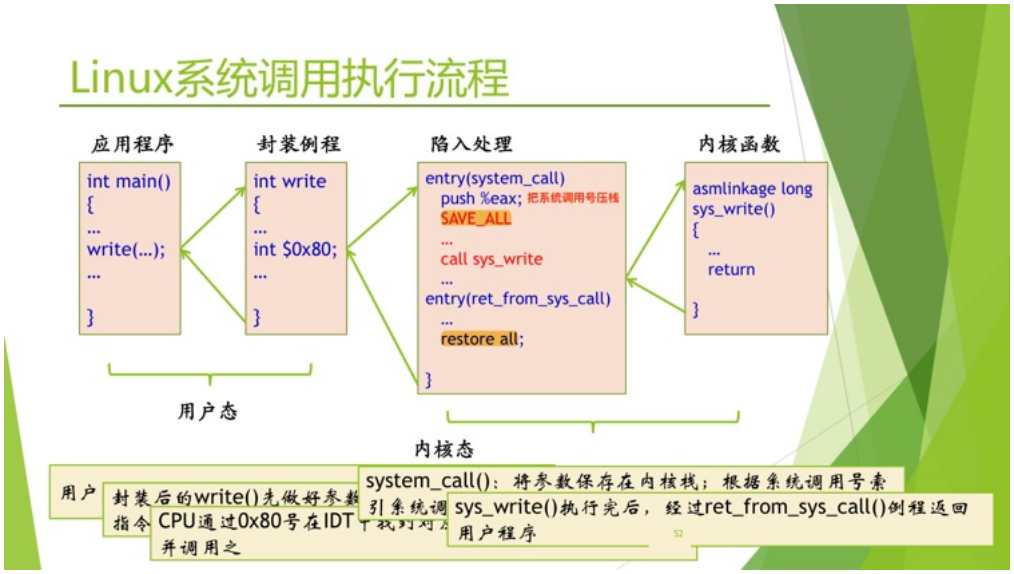
1. 由于特权级变化，硬件从 TSS 中取新的栈指针 `SS:ESP`，切换到内核栈；
2. 把用户的 `SS:ESP`、`EFLAGS`、用户态 `CS:EIP` 依次压入内核栈；
3. 把 EFLAGS 中 `TF` 位复位（关单步），`IF` 是否清零取决于门类型：陷阱门**不**自动关，中断门会关；
4. 用 128 在 IDT 中找到陷阱门描述符，从中取出段选择符放入 `CS`；
5. 段描述符基地址 + 陷阱门描述符偏移 -> `system_call` 入口地址。



图示：特权级检查 + 系统调用号 / 参数寄存器约定。

1. 进行特权级检查：**代码只能访问相同或较低特权级的数据**。
2. 系统调用号、参数仍保存在用户态使用过的寄存器中（`eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`）。

2.10.4 `system_call` 的执行流程



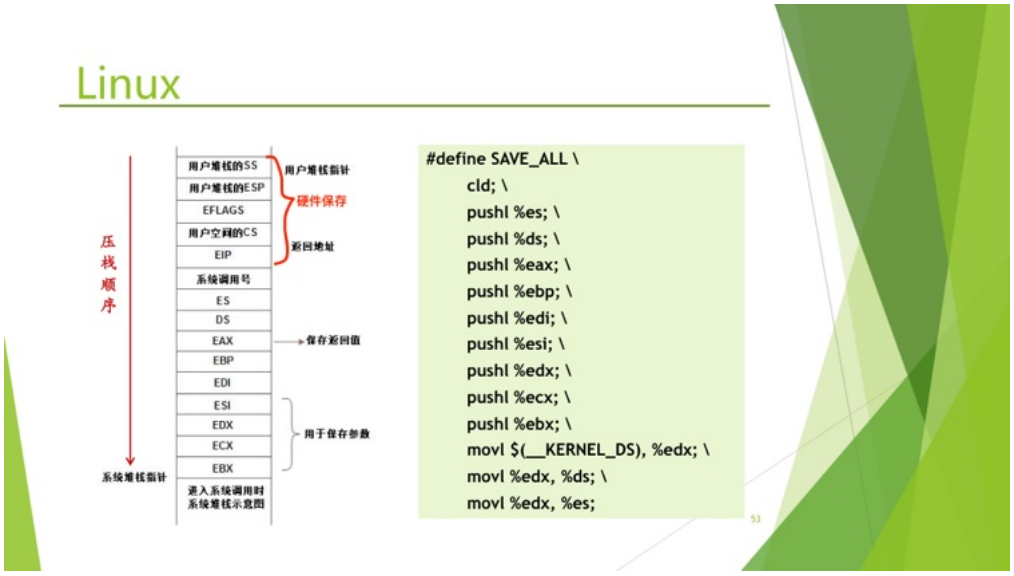
图示：用户态 `write()` -> 封装例程 -> `int $0x80` -> 内核 `system_call()` -> `sys_write` -> `ret_from_sys_call` -> 用户态。

执行流程文字版：

1. 用户态调用 C 库函数（例如 `write()`）；
2. 封装例程把参数放进对应寄存器，执行 `int $0x80`；
3. CPU 通过 0x80 在 IDT 找到 `system_call`，跳转到内核；
4. `system_call`：
   - `pushl %eax` 把系统调用号压栈；
   - `SAVE_ALL` 保存所有用户寄存器；
   - 检查 `eax` 是否合法系统调用号；
   - 通过 `sys_call_table` 调用对应内核函数（如 `sys_write`）；
   - 把返回值写回栈中 `eax`；
5. 进入 `ret_from_sys_call`：检查重新调度需求、未处理信号；
6. `RESTORE_ALL` 恢复寄存器；
7. `iret` 返回用户态原指令之后。

2.10.5 `SAVE_ALL` 宏

```
#define SAVE_ALL \
    cld;                    \
    pushl %es;              \
    pushl %ds;              \
```



图示：依次压入段寄存器、eax、ebp、edi、esi、edx、ecx、ebx，并切换到内核数据段。

```
5     pushl %eax;                   \
6     pushl %ebp;                   \
7     pushl %edi;                   \
8     pushl %esi;                   \
9     pushl %edx;                   \
10    pushl %ecx;                   \
11    pushl %ebx;                   \
12    movl  $(__KERNEL_DS), %edx;   \
13    movl  %edx, %ds;              \
14    movl  %edx, %es;
```

- `cld` 清除方向标志，保证字符串操作向高地址。
- 段寄存器和通用寄存器按固定顺序入栈，便于后续 `RESTORE_ALL` 严格对称恢复。
- `__KERNEL_DS` 是内核数据段选择符，把 `ds`、`es` 切到内核数据段，避免使用用户段进行内核访问。
- 这一段就是“**保护现场**”的实现。

2.10.6 `sys_call_table` 与系统调用分派

```
1  .data
2  ENTRY(sys_call_table)
3      .long SYMBOL_NAME(sys_exit)
4      .long SYMBOL_NAME(sys_fork)
5      .long SYMBOL_NAME(sys_read)
6      .long SYMBOL_NAME(sys_write)
7      /* ... */
```

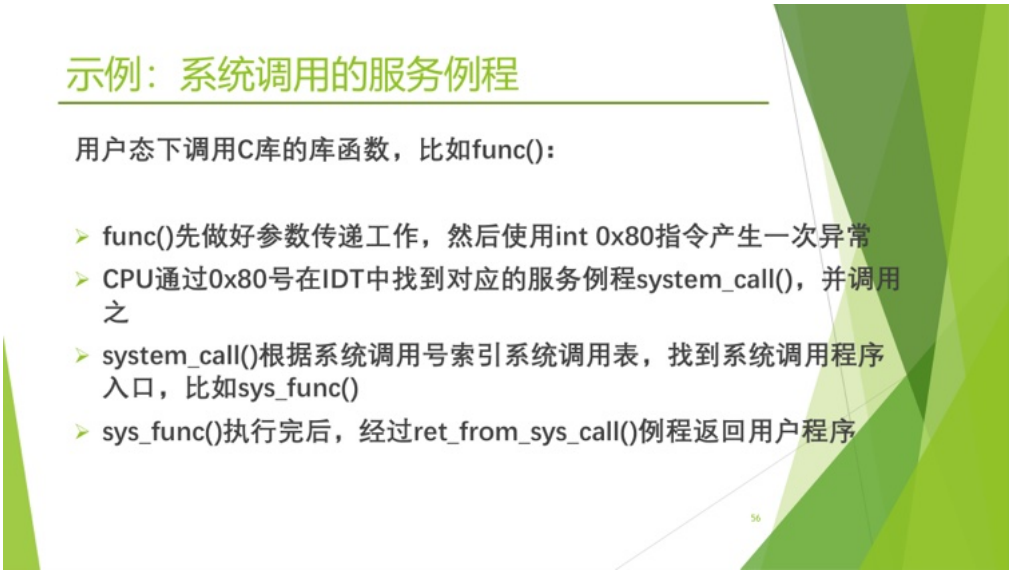
- 该表按系统调用号索引；32 位系统每项 4 字节。



图示：entry.S 中通过 .long SYMBOL_NAME(sys_xxx) 注册每个系统调用入口。

- 调用时通过 call *sys_call_table(,%eax,4) 实现“按号分派”。
- 增加新系统调用要做三件事：在 unistd.h 加号、在内核实现 sys_xxx、在 sys_call_table 中插入条目。

2.10.7 系统调用服务例程示例



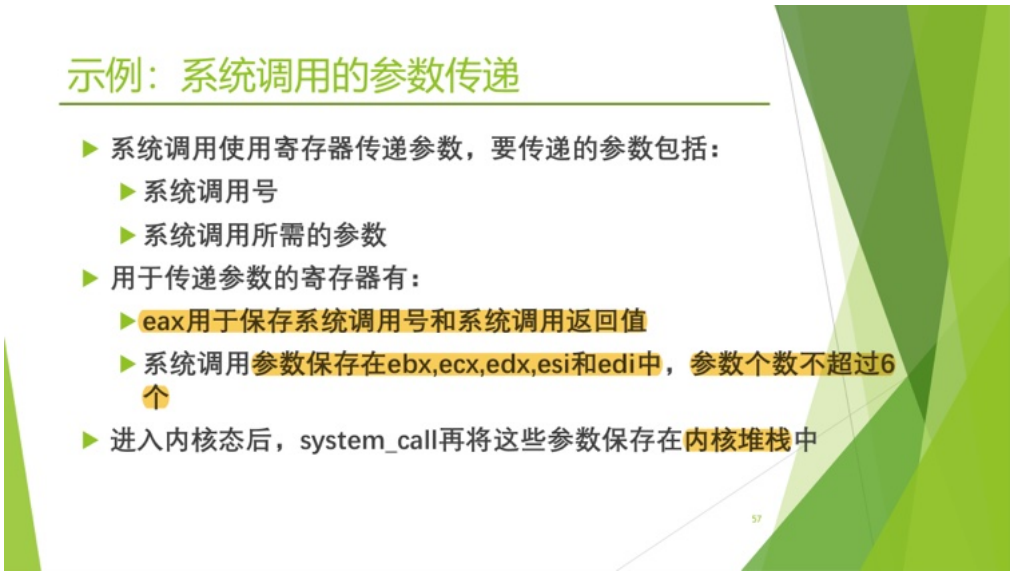
图示：用户调用 func() -> int $0x80 -> system_call() -> 查 sys_call_table -> sys_func() -> ret_from_sys_call -> 用户态。

完整流程：

1. 用户态执行 C 库 func()；
2. func() 准备参数并执行 int $0x80；
3. CPU 通过 0x80 找到 system_call；
4. system_call 根据系统调用号索引 sys_call_table 找到 sys_func；

5. 执行 sys_func，结果放入 eax；
6. ret_from_sys_call 恢复后返回用户态。

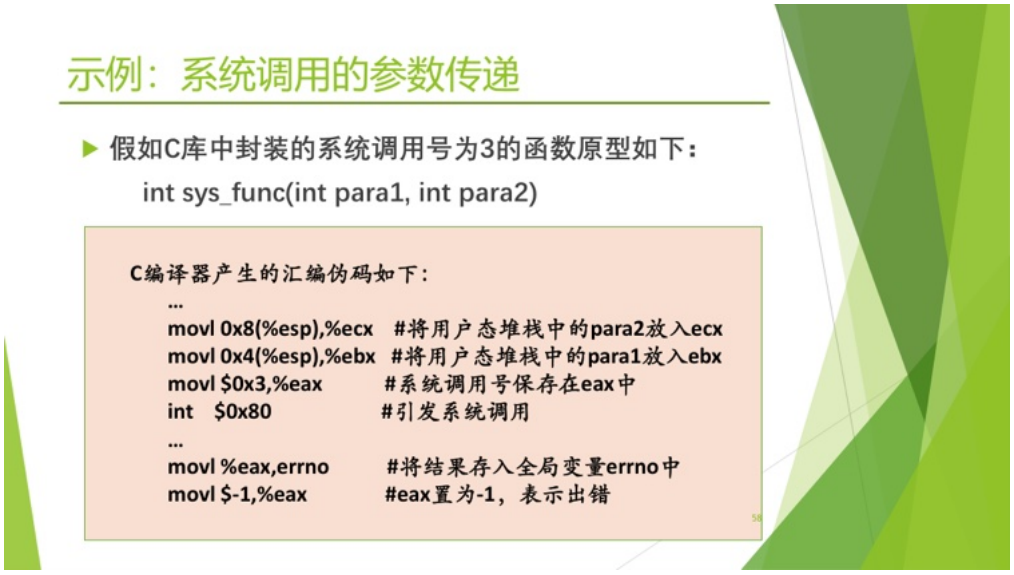
2.10.8 系统调用的参数传递（汇编视角）



图示：eax 保存系统调用号 / 返回值；ebx，ecx，edx，esi，edi 保存最多 5 个参数。

约定：

- eax <- 系统调用号；返回时 eax 保存返回值。
- ebx，ecx，edx，esi，edi（必要时含 ebp）<- 最多 6 个参数。
- 进入内核后 system_call 再把这些寄存器压栈，使内核函数可以像普通 C 函数那样从栈上读参数。



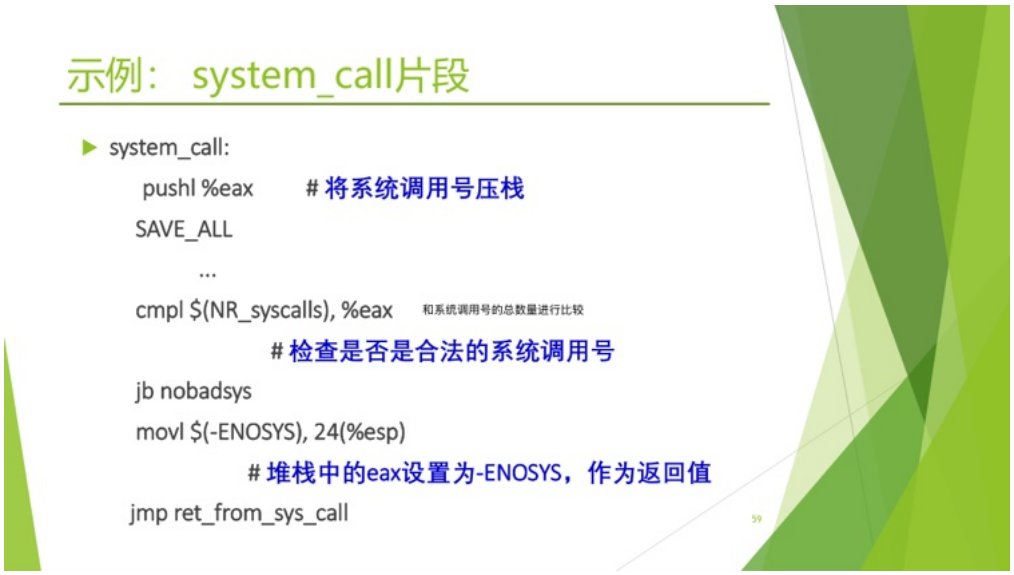
图示：int sys_func(int para1, int para2) 在用户态被调用时的汇编传参过程。

C 库封装中，对于 int sys_func(int para1, int para2)：

```
movl 0x8(%esp), %ecx   # 将用户栈中的 para2 -> ecx
movl 0x4(%esp), %ebx   # 将用户栈中的 para1 -> ebx
movl $0x3, %eax        # 系统调用号 = 3
int  $0x80             # 引发系统调用
movl %eax, errno       # 出错时的返回值写到 errno
movl $-1, %eax         # 失败时把 -1 作为返回值
```

调用时寄存器值：eax = 3，ebx = para1，ecx = para2。

2.10.9 system_call 关键片段

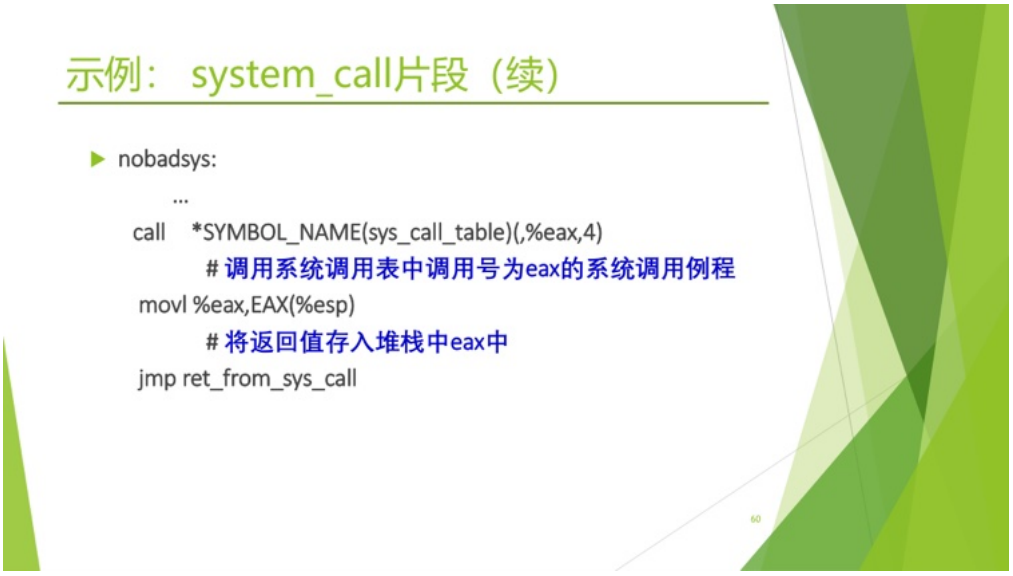


图示：pushl %eax 把系统调用号压栈、SAVE_ALL、合法性检查。

```
system_call:
    pushl %eax                  # 把系统调用号压栈
    SAVE_ALL
    ...
    cmpl  $(NR_syscalls), %eax # 检查是否合法系统调用号
    jb    nobadsys
    movl  $(-ENOSYS), 24(%esp) # 非法 -> 把 eax 在栈中的位置写为 -ENOSYS
    jmp   ret_from_sys_call
```

- NR_syscalls 是系统调用总数。
- 若 eax >= NR_syscalls，把栈中保存的 eax 改为 -ENOSYS 作为返回值，跳到返回路径。

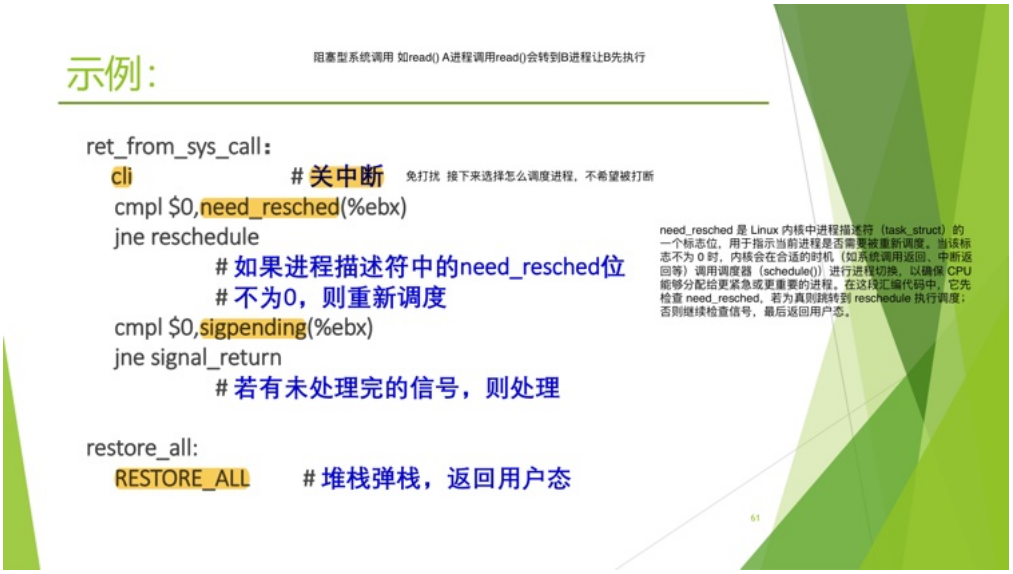
```
nobadsys:
    ...
    call  *SYMBOL_NAME(sys_call_table)(,%eax,4)
    movl  %eax, EAX(%esp)   # 把返回值写回栈中 eax 的位置
    jmp   ret_from_sys_call
```



图示：`call *sys_call_table(,%eax,4)` 按号分派；返回值写回栈中 `eax`。

- 这就是“按号分派”的关键一步。
- 把返回值放回内核栈中 `eax` 对应的位置，是为了让后续 `RESTORE_ALL` 把返回值带回用户态 `eax`。

2.10.10 `ret_from_sys_call`



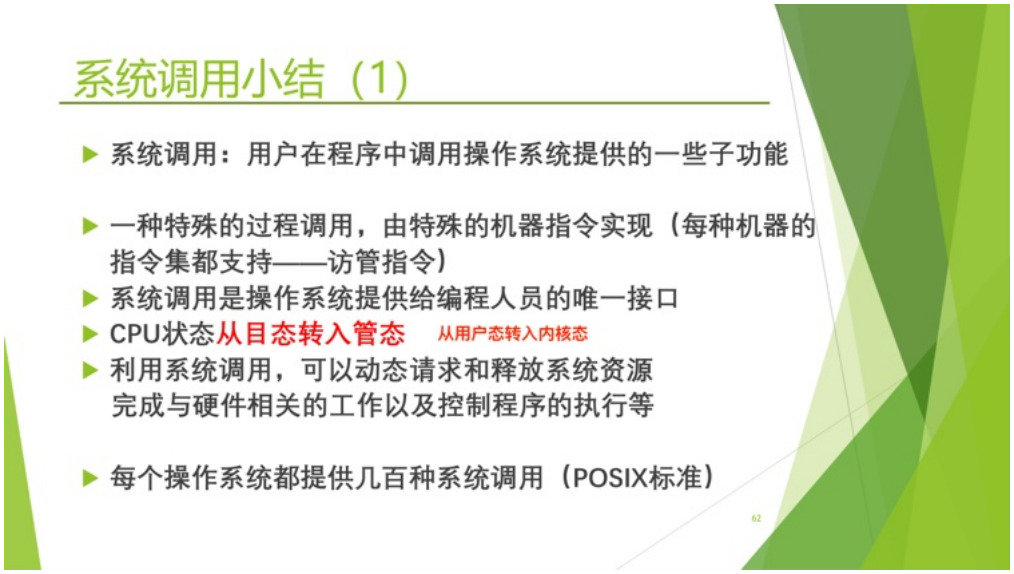
图示：检查 `need_resched` 决定是否重新调度，检查 `sigpending` 决定是否处理信号，最后 `RESTORE_ALL` 恢复并返回用户态。

```
ret_from_sys_call:
    cli
    cmpl $0, need_resched(%ebx)
    jne  reschedule          # 若需要重新调度，跳到调度路径
    cmpl $0, sigpending(%ebx)
    jne  signal_return       # 若有未处理信号，跳到信号处理
restore_all:
```

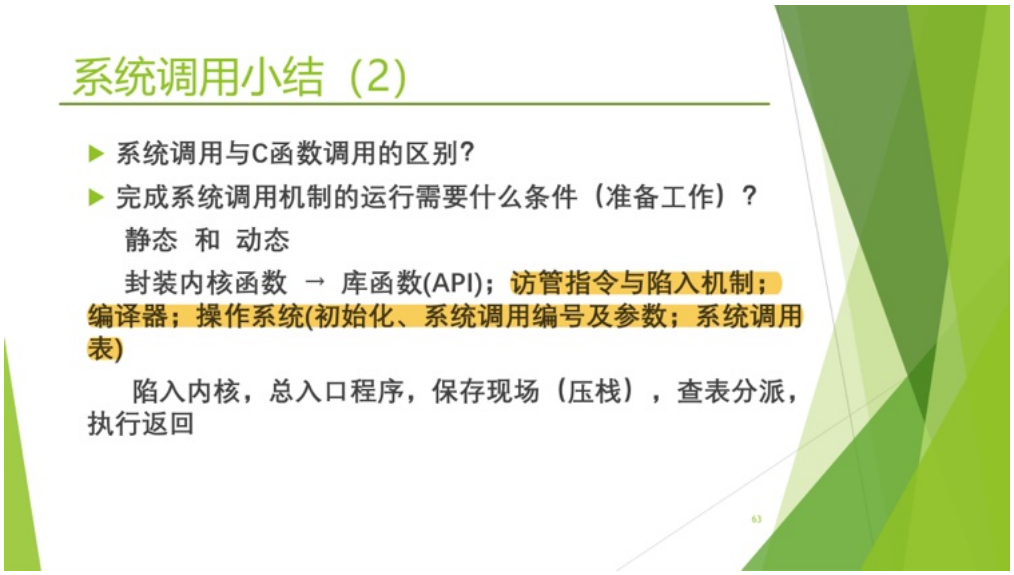
```
8       RESTORE_ALL             # 弹栈恢复寄存器，iret 返回用户态
```

- `need_resched` 是 `task_struct` 中的标志，表示当前进程是否需要被重新调度；该标志被时钟中断、`wake_up`、`yield` 等设置。
- `sigpending` 表示是否有待处理信号；若有，进入信号处理路径，可能在用户栈构造信号上下文。
- `cli` 关中断是为了避免在重新调度 / 信号处理过程中被打断；这里短暂关中断不会破坏整体并发性。

2.10.11 系统调用小结



图示：系统调用是一种特殊的过程调用；用户态进入内核态；几百种系统调用对应 POSIX 标准。



图示：完成系统调用机制需要静态准备（封装例程、IDT 表项、系统调用表）和动态过程（陷入、保存、查表、分派、返回）。

- **系统调用与 C 函数调用的区别**：见 §8.2。
- **系统调用机制的运行条件**：
  - **静态准备**：封装内核函数为库函数（API）；选定访管指令；设置 IDT；设置系统调用号与系统调用表；编译器配合。
  - **动态执行**：陷入内核 -> 总入口 -> 保存现场（压栈）-> 查表分派 -> 执行 -> 恢复 -> 返回。

2.11 中断发生后操作系统底层的 8 步



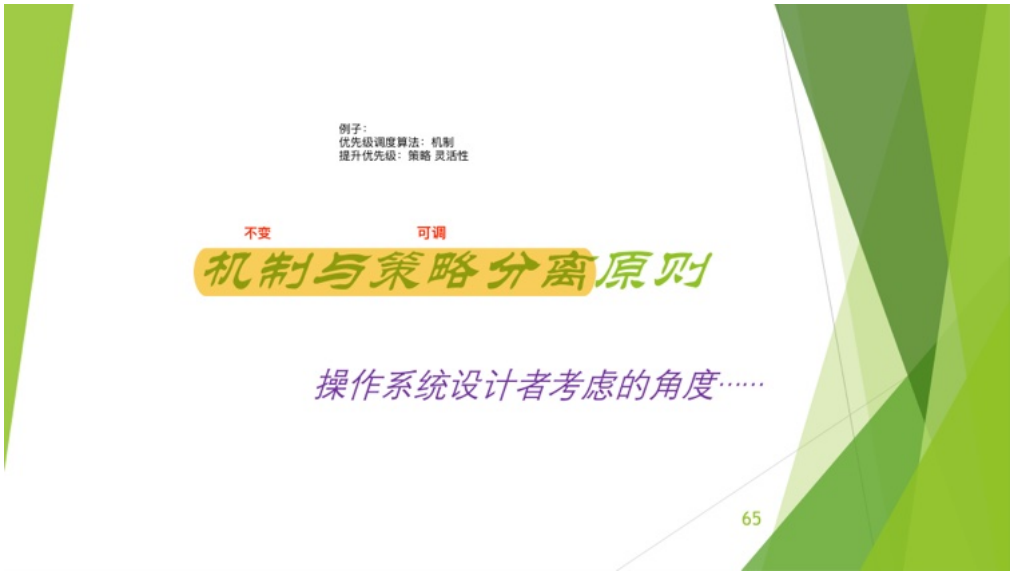
图示：从硬件压栈到调度切换到新进程的全过程。

1. **硬件压栈**：保存程序计数器等。
2. **硬件按中断向量装入新的 PC 等**。
3. **汇编入口**：保存其他寄存器值。
4. **汇编入口**：设置新的内核栈。
5. **C 语言中断服务程序**：执行（如读并缓冲输入）。
6. **进程调度**：决定下一个将运行的进程。
7. **C 语言中断处理返回**至汇编代码。
8. **汇编入口**：开始运行新的当前进程。

> 这个 8 步序列把中断、调度和上下文切换连成一条因果链，是后面进程线程调度章节的引子。

2.12 机制与策略分离原则

- **机制（Mechanism）**：实现某种能力的基础设施，要稳定、通用、不变。例如系统调用入口、中断分派、调度框架。
- **策略（Policy）**：在机制之上做选择，决定“怎样做”。例如调度算法、页面置换算法、I/O 缓



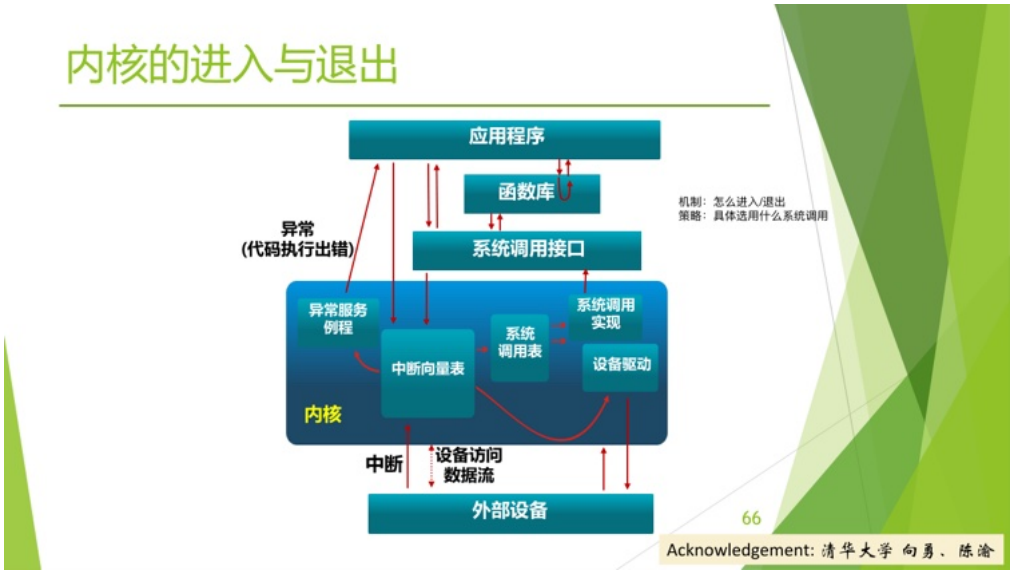
图示：机制保持稳定提供基础能力；策略灵活替换决定具体行为。

冲策略。

- 该原则带来两点收益：
  - 机制相对稳定，便于硬件抽象和长期维护；
  - 策略可灵活替换，便于针对不同场景优化。

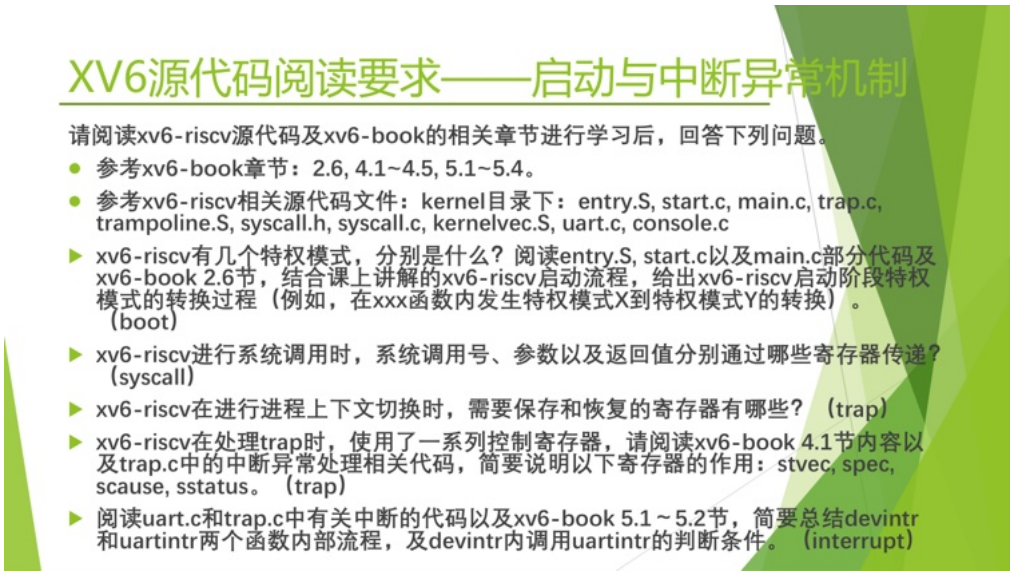
例子：

- **优先级调度**中的“维护多级队列、按优先级取队列头”是机制；“如何提升或降低优先级”是策略；
- **系统调用**中的“陷入入口、按号分派、统一返回”是机制；具体每个 `sys_xxx` 实现是策略。



图示：中断、异常、系统调用都通过统一入口进入内核，再通过设备驱动或系统调用实现完成具体功能。

2.13 XV6 源代码阅读题（来自第 67 页）



图示：xv6-book 第 2.6、4.1 ~ 4.5、5.1 ~ 5.4 节，配套 `kernel/` 目录下的若干文件，是本节阅读题的素材范围。

> **代码阅读题：** xv6-riscv 有几个特权模式？分别是什么？阅读 `entry.S`、`start.c` 和 `main.c` 的部分代码以及 xv6-book 2.6 节，结合课上讲解，给出 xv6-riscv 启动阶段特权模式的转换过程。

解答：

- xv6-riscv 涉及 RISC-V 三种特权模式：**机器态（M-mode）**、**监管态（S-mode）**、**用户态（U-mode）**。
- 启动每个 hart 的总流程：
  1. **`_entry`（M-mode，`entry.S`）**：每个 hart 选自己的内核栈（按 hart id 偏移），跳到 C 函数 `start`。
  2. **`start`（M-mode，`start.c`）**：仍在 M-mode 执行。主要工作：
     - 把 `mstatus.MPP` 设为 S-mode，使下一次 `mret` 后进入 S-mode；
     - 设置 `mepc = main`，让 `mret` 之后跳到 `main`；
     - 把异常 / 中断委托给 S-mode：`medeleg`、`mideleg` 都置全；
     - 配置 PMP，使 S-mode 能访问全部物理地址（`pmpaddr0`、`pmpcfg0`）；
     - 启用时钟中断并配置时钟相关 MSR；
     - 把 hart id 写入 `tp` 寄存器；
     - 执行 `mret`：由 M-mode 切到 S-mode，并跳到 `main`。
  3. **`main`（S-mode，`main.c`）**：完成页表、中断、进程、文件系统等初始化；只有 hart 0 做全局初始化，其他核心等待并随后进入调度循环。
  4. **`userinit` / `usertrapret`**：通过 `sret` 从 S-mode 切到 U-mode，执行第一个用户进程（initcode）。

- 总结：
  - **M-mode -> S-mode** 的切换发生在 `start` 末尾的 `mret`；
  - **S-mode -> U-mode** 的切换发生在 `usertrapret` 中的 `sret`；
  - 用户态系统调用 / 异常通过 `ecall` 触发 `uservec`，先回到 S-mode 内核继续处理。

> **代码阅读题：** xv6-riscv 中断 / 异常时保存了哪些寄存器？

解答：

- 关注三处源码：
  - **`kernelvec.S`（内核态 -> 内核态陷入入口）**：在内核栈上保存所有调用者保存寄存器（`ra`，`sp`，`gp`，`tp`，`t0~t6`，`a0~a7`，`s0~s11`），调用 `kerneltrap`，再恢复并 `sret`。
  - **`trampoline.S` / `uservec`（用户态 -> 内核态陷入入口）**：利用 `sscratch` 中存放的进程 `trapframe` 地址，把用户全部通用寄存器保存到 `trapframe` 中（`ra`、`sp`、`gp`、`tp`、`t0~t6`、`s0~s11`、`a0~a7`），同时保存 `kernel_sp`、`kernel_satp`、`kernel_trap`、`epc`、`hartid` 等以便后续切换页表与栈。
  - **`syscall.c`**：从 `trapframe` 中读 `a7` 作为系统调用号、`a0~a5` 作为参数，分派到具体 `sys_*`；返回值写回 `trapframe.a0`。
- 设计要点：
  - 用户态进入内核必须保存**所有**通用寄存器，因为内核可能任意修改它们；
  - 用户态进入内核还要切换页表（`satp`）和栈（`sp`），因此需要保存 `kernel_satp` 和 `kernel_sp`；
  - 内核态陷入只需保存调用者保存寄存器，因为调用约定保证被调用者保存寄存器（`s0~s11`）由被调用方负责。

2.14 本讲小结

- 操作系统是事件驱动的，事件经由中断/异常机制进入内核。
- 中断/异常机制是硬件与软件的协作产物：硬件做最少必要现场保存与跳转，软件做识别、处理与恢复。
- 系统调用是中断/异常机制的特例：统一陷入入口、按系统调用号分派、统一返回。
- IA-32 与 RISC-V 在具体寄存器和指令上不同，但抽象一致：特权级、向量分派、栈切换、参数约定。
- “机制与策略分离”是 OS 设计的普遍原则，本讲是这条原则最早出现的具体场景之一。
- 学完本讲后，应当能在脑中复述：
  - 用户态 -> 内核态的唯一路径；
  - `int $0x80` 后硬件做了什么、软件做了什么；
  - `system_call`、`sys_call_table`、`ret_from_sys_call` 各自的角色；

- 中断处理上半部 / 下半部为什么这样切；
- xv6-riscv 启动阶段的 M -> S -> U 转换链路。

LECTURE 3

第 3 讲

进程线程模型

Operating Systems Course Notes

3.1 本讲导读

- 这一讲建立操作系统中“**执行实体**”的抽象：
  - **进程（Process）**：对 CPU 的抽象 + 资源分配的独立单位；
  - **线程（Thread）**：对 CPU 调度单位的进一步细化；
  - **协程（Coroutine）/ 纤程（Fiber）**：在用户态下进一步把执行流切碎。
- 知识脉络：
  1. 从顺序程序到并发程序，再到进程；
  2. 进程的状态、PCB、地址空间、队列；
  3. 进程控制操作及 UNIX `fork / exec / wait / exit` 模型；
  4. 引入线程的动机、属性、用户级 / 核心级 / 混合实现；
  5. 协程与纤程；
  6. 与中断/异常机制的关联，以及机制 / 策略分离原则在本讲的体现。

> **三句口诀：**进程是对 CPU 的抽象；地址空间是对内存的抽象；文件系统是对磁盘的抽象。



图示：进程是对 CPU 的抽象，地址空间是对内存的抽象，文件系统是对磁盘的抽象。

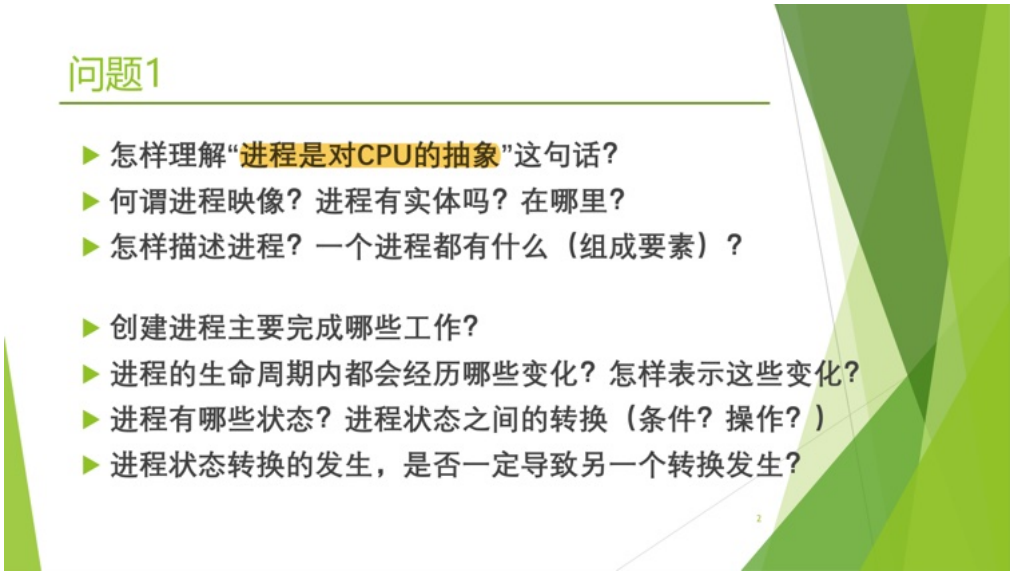
3.2 三组开篇思考题

3.2.1 怎样理解“进程是对 CPU 的抽象”？

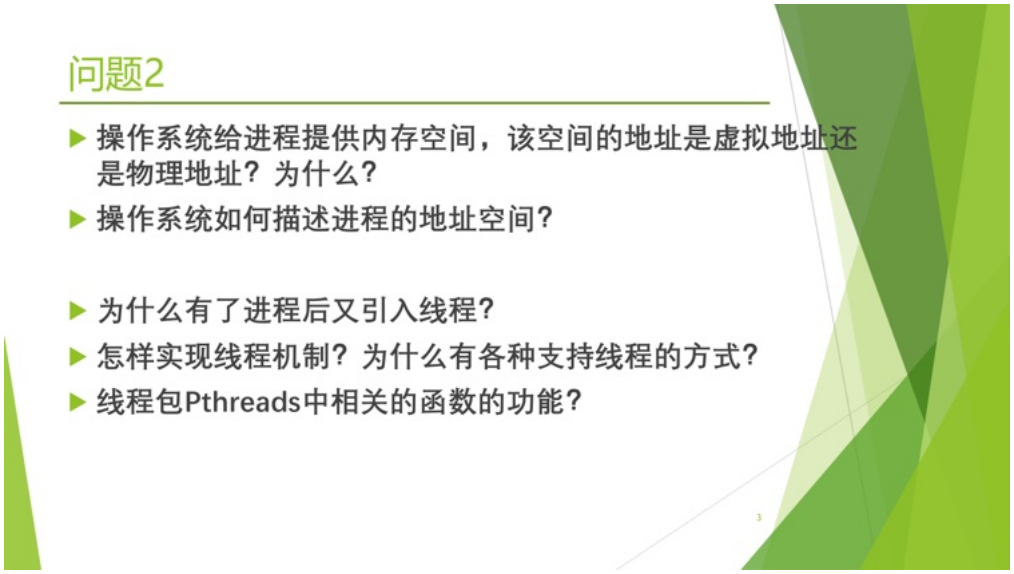
> **思考题：**怎样理解“进程是对 CPU 的抽象”?

解答：

- 物理 CPU 数量有限，但系统中可以同时存在大量执行实体。



图示：进程的概念、组成、生命周期与状态转换的开篇问题。



图示：地址空间、线程引入及实现方式相关问题。

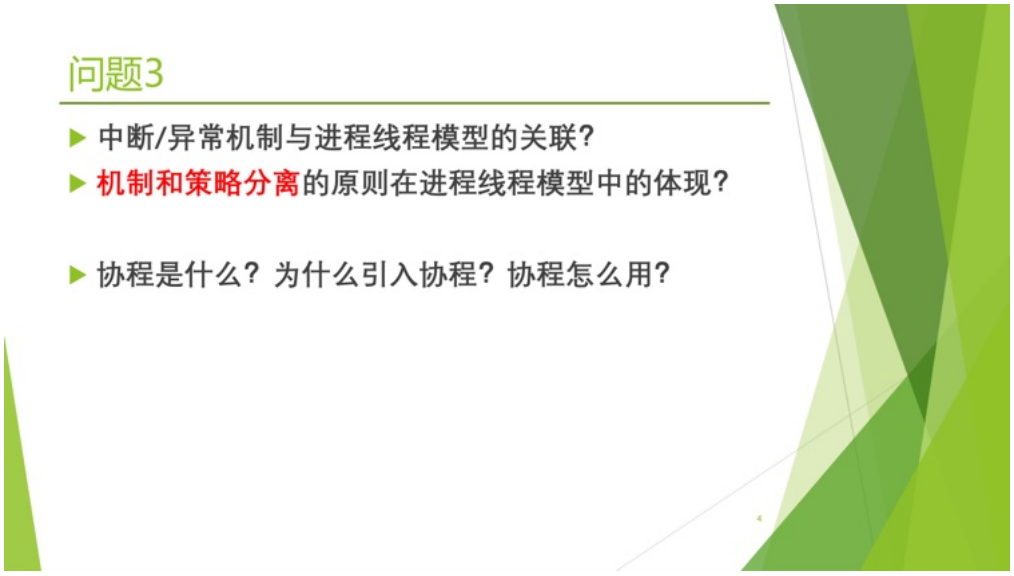
- 操作系统通过多道程序设计 + 调度，让不同执行实体在 CPU 上轮流运行，使每个执行实体都觉得自己拥有一个独占的 CPU。
- 每个进程都有独立的执行上下文（PC、寄存器、栈），所以即使被打断，也能从断点继续。
- 把这一组“**逻辑独立、互不干扰**”的执行单位称为进程，就把“**有限的物理 CPU**”抽象成了“**多个虚拟 CPU**”。

3.2.2 进程映像与进程实体

> **思考题：**进程映像是什么? 进程有实体吗? 在哪里?

解答：

- **进程映像（Process Image）**是一个进程在内存中所有可见状态的集合，包括：



图示：进程线程模型与中断异常机制、机制策略分离、协程的关联。

  - 程序代码段（`.text`）；
  - 数据段（`.data`、`.bss`）；
  - 堆（heap）；
  - 栈（stack）；
  - 共享库映射；
  - 进程控制块（PCB）。
- **进程实体存在**：
  - 用户地址空间部分（代码、数据、堆、栈、共享库）位于内存；
  - PCB 等管理结构位于内核空间；
  - 当进程被挂起或换出时，部分映像可能保存在磁盘的交换分区。
- 一个进程的“身份”由其 PCB 和地址空间共同确定。

3.2.3 进程的组成要素

> **思考题：**怎样描述进程？一个进程都有什么？

解答：

一个进程由四类信息构成：

1. **程序代码**：可执行指令序列。
2. **数据集合**：全局变量、静态变量、动态分配内存。
3. **执行上下文**：CPU 寄存器、PC、PSW、栈指针等。
4. **进程控制块（PCB）**：操作系统对该进程的管理记录。

OS 通过 PCB 来感知和管理进程；PCB 与进程是一一对应的。

3.2.4 创建进程主要做哪些工作？

> **思考题：**创建进程主要完成哪些工作?

解答：

1. 分配唯一的进程标识符 PID 和进程控制块 PCB；
2. 为进程分配地址空间（复制父进程或加载新可执行文件）；
3. 初始化 PCB，设置默认值（如初始状态、优先级、队列指针）；
4. 加载代码和数据到地址空间；
5. 初始化执行上下文（PC、栈指针、寄存器）；
6. 分配资源（文件描述符、信号处理状态等）；
7. 把 PCB 插入就绪队列等待调度。

3.2.5 进程生命周期与状态转换

> **思考题：**进程生命周期内会经历哪些变化？怎样表示?

解答：

- 主要状态：**创建（New）-> 就绪（Ready）-> 运行（Running）-> 等待 / 阻塞（Waiting）-> 终止（Terminated）**，可加入挂起态。
- 通常用**进程状态转换图**表示，节点是状态，边是触发条件。
- 见 §3 详细模型。

3.2.6 状态转换是否会引发其他转换？

> **思考题：**进程状态转换的发生，是否一定导致另一个转换发生?

解答：

是的。OS 是动态系统，状态转换通常成对发生：

- 一个进程 `Running -> Waiting`，立即释放 CPU，调度器会选另一个进程使其 `Ready -> Running`。
- 一个进程 `Waiting -> Ready`，可能在未来导致它 `Ready -> Running`。
- 一个进程 `Running -> Terminated`，释放资源后可能让等待该资源的进程 `Waiting -> Ready`。
- 一个进程 `Running -> Ready`（时间片到 / 被抢占），同时另一个 `Ready -> Running`。

3.2.7 进程地址空间是虚拟还是物理？

> **思考题：** OS 给进程提供的内存空间地址是虚拟还是物理？为什么？

解答： 是**虚拟地址空间**。理由：

1. **隔离与保护**：每个进程独立空间，互不干扰；
2. **空间扩展**：虚拟空间可大于物理内存，借助 MMU 与磁盘交换；
3. **简化内存管理**：物理上不连续的页可以映射到虚拟上连续的范围；
4. **简化加载与链接**：编译期就能确定虚拟布局，不依赖物理装载位置；
5. **支持共享**：通过把不同进程的虚拟页映射到同一物理页，实现代码或数据共享。

3.2.8 OS 如何描述地址空间？

> **思考题：** 操作系统如何描述进程的地址空间？

解答：

- Linux 使用 `mm_struct` 表示一个进程的整个地址空间；
- `mm_struct` 内部维护一组 `vm_area_struct`（VMA），每个 VMA 表示一段连续区域（起止地址、权限、是否映射到文件等）；
- 通过页表把虚拟地址翻译成物理地址，页表由 OS 维护，硬件 MMU 使用；
- 用户态可通过 `cat /proc/<pid>/maps`、`pmap` 查看；
- XV6 使用 `proc->pagetable`，用 `vmprint` 等工具打印。

3.2.9 为什么需要线程

> **思考题：** 有了进程为什么还要线程？

解答：

- 进程作为资源分配单位创建/切换开销大；
- 应用程序内部经常需要多个执行流并发推进（字处理、Web 服务器、GUI）；
- 同一进程内多线程共享地址空间和资源，通信效率高；
- 多核 CPU 上多线程可真正并行；
- 线程因此被称为“轻量级进程”。

3.2.10 线程怎么实现，为何有多种方式

> **思考题：**怎样实现线程机制？为什么有多种方式？

解答：

- **用户级线程（ULT）**：库实现，内核无感；切换快但阻塞影响整个进程，不能利用多核。
- **核心级线程（KLT）**：内核实现；阻塞不影响其他线程，可利用多核；切换需进入内核。
- **混合实现**：用户级线程映射到内核级线程（M:N），兼顾两者优势但实现复杂。

存在多种方式是“开销 / 并发 / 可扩展性”之间的折中。

3.2.11 Pthreads 主要函数

> **思考题：** Pthreads 提供了哪些函数？

解答：

函数	作用
`pthread_create()`	创建新线程
`pthread_exit()`	终止当前线程
`pthread_join()`	等待指定线程结束
`pthread_yield()`	主动让出 CPU
`pthread_self()`	取自身线程 ID
`pthread_mutex_*`	互斥锁初始化 / 加锁 / 解锁 / 销毁
`pthread_cond_*`	条件变量初始化 / 等待 / 唤醒 / 销毁

> **要点：**线程通常**自愿**让出 CPU；POSIX 没有强制抢占式语义，调度细节交给具体实现。

3.2.12 中断 / 异常机制与进程线程模型的关联

> **思考题：**中断/异常机制和进程线程模型有什么关联？

解答：

1. **时钟中断**触发调度器抢占当前进程，把 `Running` 进程切回 `Ready`。
2. **I/O 完成中断**把等待该设备的进程从 `Waiting` 转到 `Ready`。
3. **系统调用**通过陷入异常进入内核，可能导致阻塞或新建/撤销进程。
4. **故障类异常**（如缺页）触发内核换页，进程会被暂停。
5. **上下文切换**在中断/异常机制下完成：硬件保存最少现场，内核保存其余上下文，再切换 PCB / 页表 / 栈。

3.2.13 机制 / 策略分离在本讲的体现

> **思考题：**机制和策略分离在进程线程模型中怎么体现？

解答：

维度	机制	策略
状态管理	阻塞 / 唤醒 / 上下文切换原语	调度算法（FCFS / 优先级 / 多级反馈队列）
挂起机制	把进程映像换到磁盘 / 换回内存	何时挂起、挂起谁（负载调节）
线程实现	用户级 / 核心级 / 混合的运行时	应用并发模型（线程池规模、任务划分）
`fork` 实现	复制 PCB、地址空间 + COW	子进程是否立即被调度、运行什么程序

3.2.14 协程是什么、为什么、怎么用

> **思考题：**协程是什么？为什么引入？怎么用？

解答：

- **协程（Coroutine）**是用户态、协作式的并发单元，由程序员在用户态显式控制切换；
- 引入原因：
  - 处理高并发 I/O（C10K、C100K 问题）时，线程数量过多导致内存与切换开销爆炸；
  - 用同步代码表达异步逻辑，避免回调地狱；
  - 切换开销极低，无需进入内核；
- 使用方式：
  - 语言原生：Python `async/await`，Go `goroutine`，Rust `async`，C++20 `co_await/co_yield`；
  - 关键模式：在 I/O 等待点 `yield` 让出，由事件循环 / 调度器恢复。

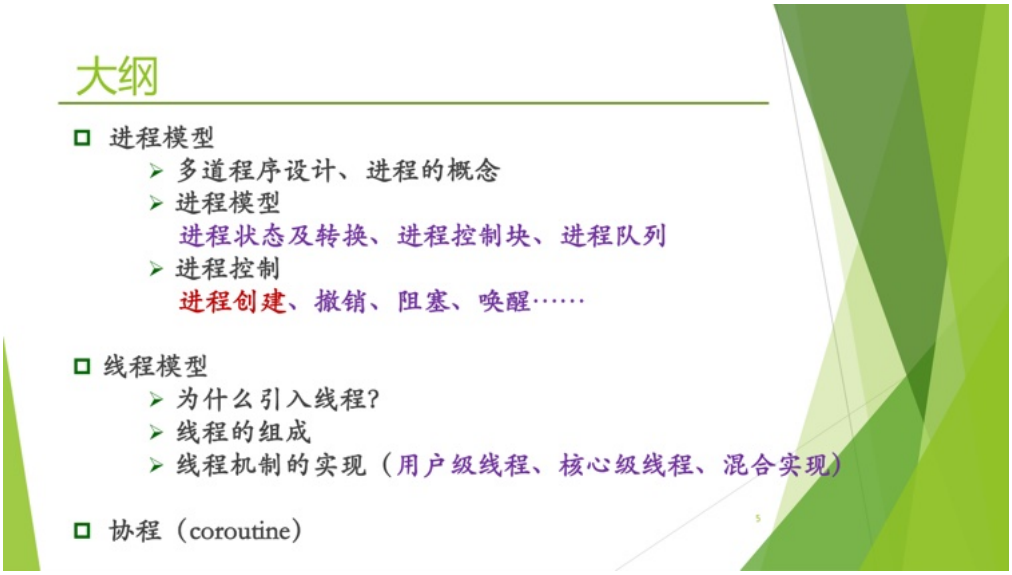
3.2.15 纤程

> **拓展：**纤程（Fiber）是 Windows 提供的一种线程内可调度的用户态执行单元，调度由用户代码控制。它比线程粒度更小，但比 POSIX 协程更接近“线程式 API”。

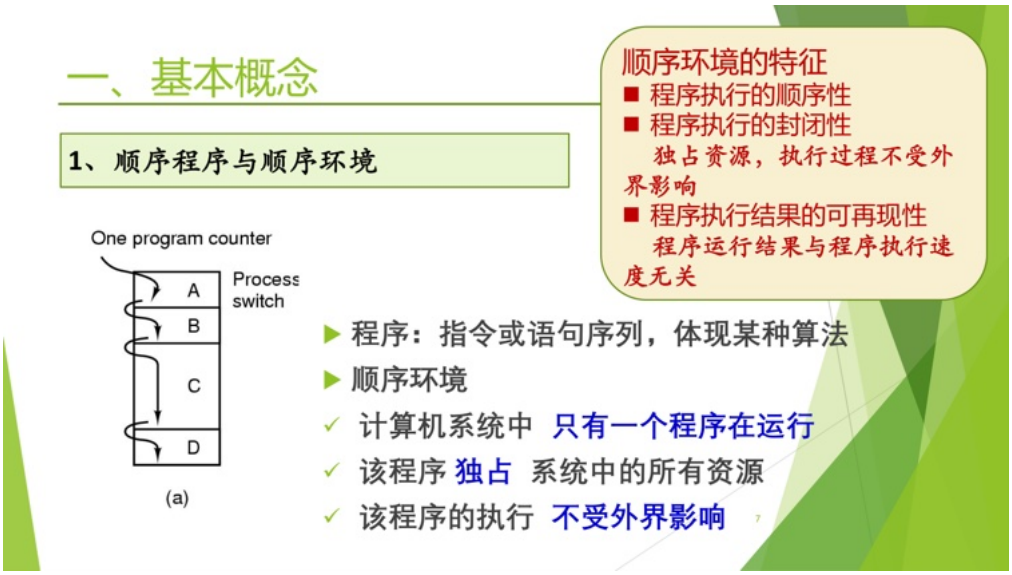
3.3 进程模型基础概念

3.3.1 顺序程序与顺序环境

- **程序（Program）**是指令或语句的序列，体现某种算法，是静态的。
- **顺序环境**指系统中只有一个程序在运行，独占资源，执行不受外界干扰。
- 顺序环境的三大特征：
  - **顺序性**：指令严格按程序顺序执行；



图示：进程模型 -> 进程控制 -> 线程模型 -> 协程，依次展开。



图示：在顺序环境下，程序独占资源，按顺序执行，结果可再现。

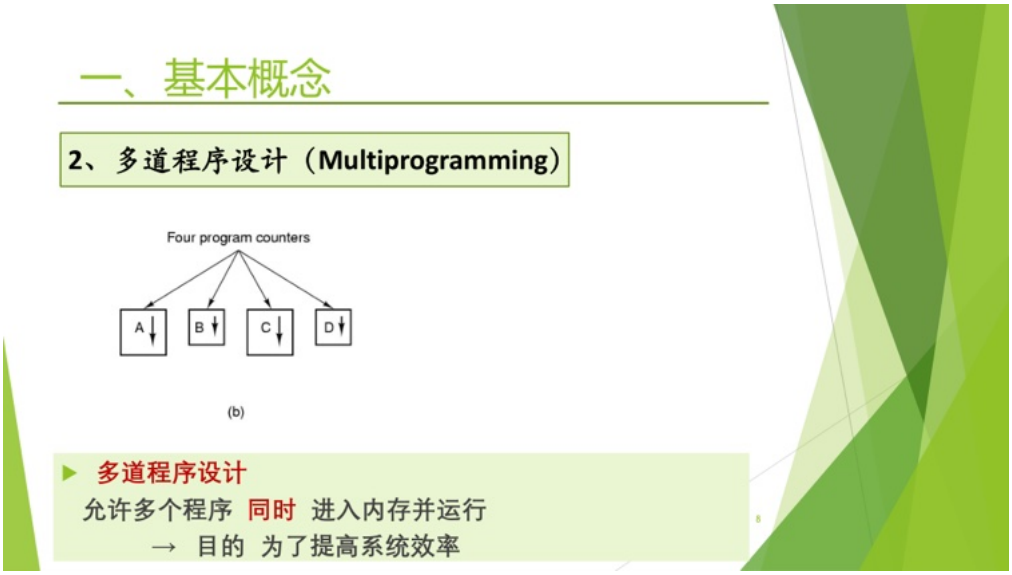
- **封闭性**：独占资源，运行不受外界影响；
- **可再现性**：相同输入下结果总相同，与执行速度无关。

3.3.2 多道程序设计

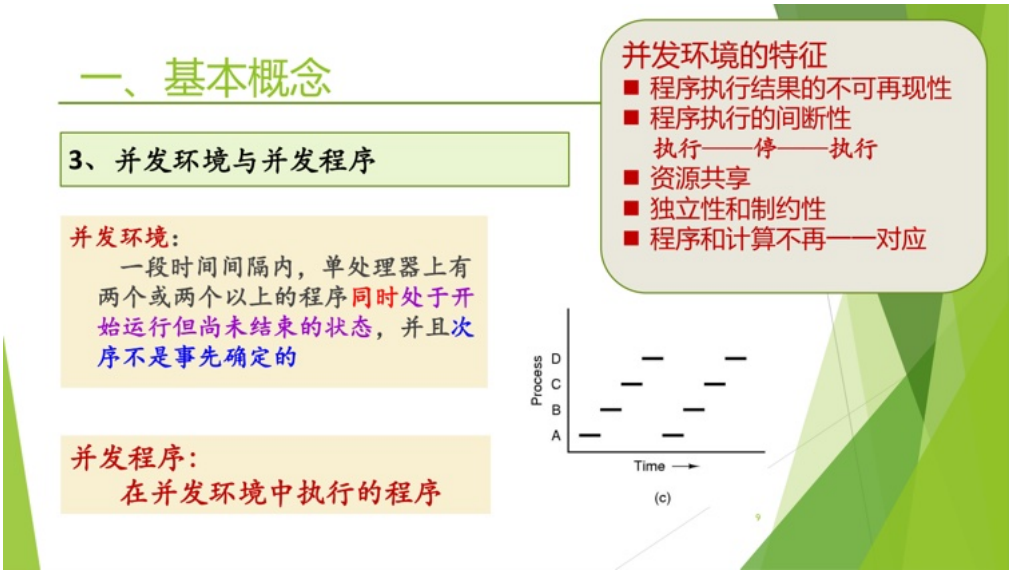
- **多道程序设计（Multiprogramming）**：允许多个程序同时进入内存并交替运行。
- 目的：当一个程序在等待 I/O 时让 CPU 切换到其他程序，提高 CPU 利用率与系统吞吐量。
- 也是后续“并发”、“调度”、“地址空间”等概念存在的前提。

3.3.3 并发环境与并发程序

- **并发环境**：在一段时间间隔内，单处理器上有两个或以上程序处于“开始运行但尚未结束”的状态，且执行次序不确定。



图示：允许多个程序同时进入内存，CPU 在它们之间切换。

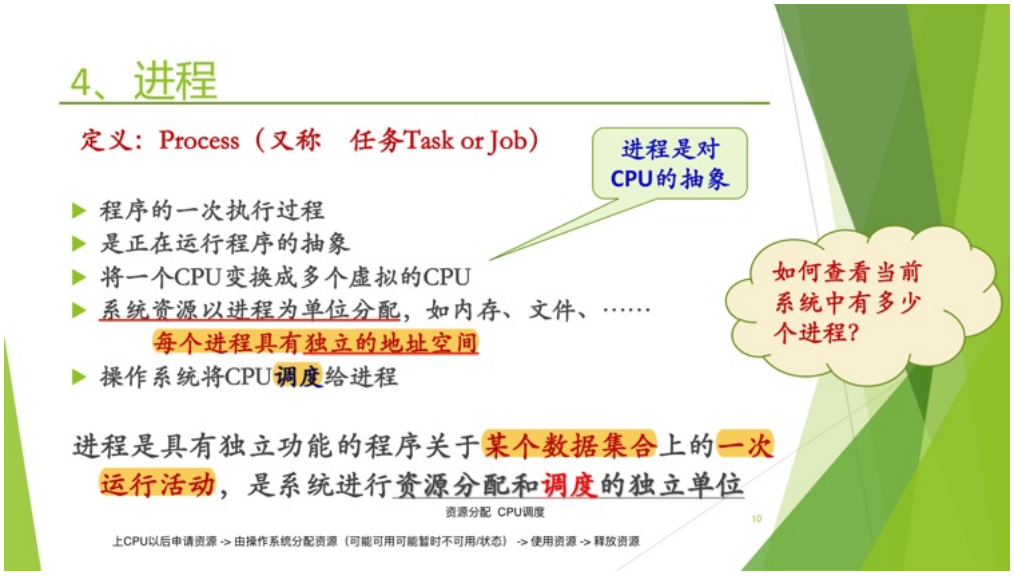


图示：并发程序在共享资源下交替推进，结果不再可再现。

- **并发程序**：在并发环境中执行的程序。
- 并发程序的特征：
  - **执行结果的不可再现性**；
  - **执行的间断性**（执行—停—执行）；
  - **资源共享**；
  - **独立性与制约性并存**；
  - **程序与计算不再一一对应**：一个程序可对应多次执行（多个进程）。

3.3.4 进程的定义

- 进程（Process），又称任务（Task）或作业（Job）。
- 多种角度的定义：



图示：进程 = 程序的一次执行过程 = 资源分配与调度的独立单位 = 对 CPU 的抽象。

  - “程序的一次执行过程”；
  - “正在运行程序的抽象”；
  - “具有独立功能的程序在某个数据集合上的一次运行活动”；
  - 是系统资源分配和调度的独立单位。
- 进程是对 CPU 的抽象：把一个或多个物理 CPU 转换为多个虚拟 CPU。
- 资源分配 vs CPU 调度：
  - 进程是资源分配单位（内存、文件等）；
  - 进程（或其线程）是 CPU 调度单位。

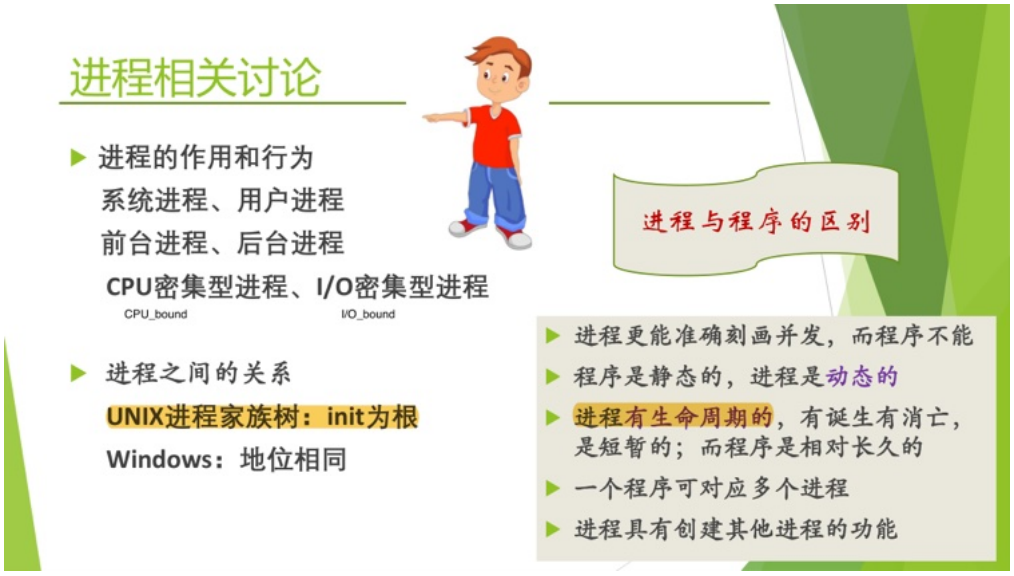
执行时的资源使用模式：

1. 上 CPU 后申请资源；
2. 由操作系统分配（可用直接给，不可用则进入等待队列）；
3. 使用资源完成功能；
4. 释放资源。

3.3.5 进程相关的若干区分

进程的多种分类：

- **系统进程 vs 用户进程**：内核任务 vs 用户应用。
- **前台进程 vs 后台进程**：与用户交互 vs 在后台默默运行。
- **CPU 密集型 vs I/O 密集型**：CPU-bound 主要消耗 CPU；I/O-bound 主要等待 I/O。
- 进程之间的关系：
  - UNIX：所有进程构成树状结构，根为 `init`；
  - Windows：进程之间地位相同，没有强制的父子树。



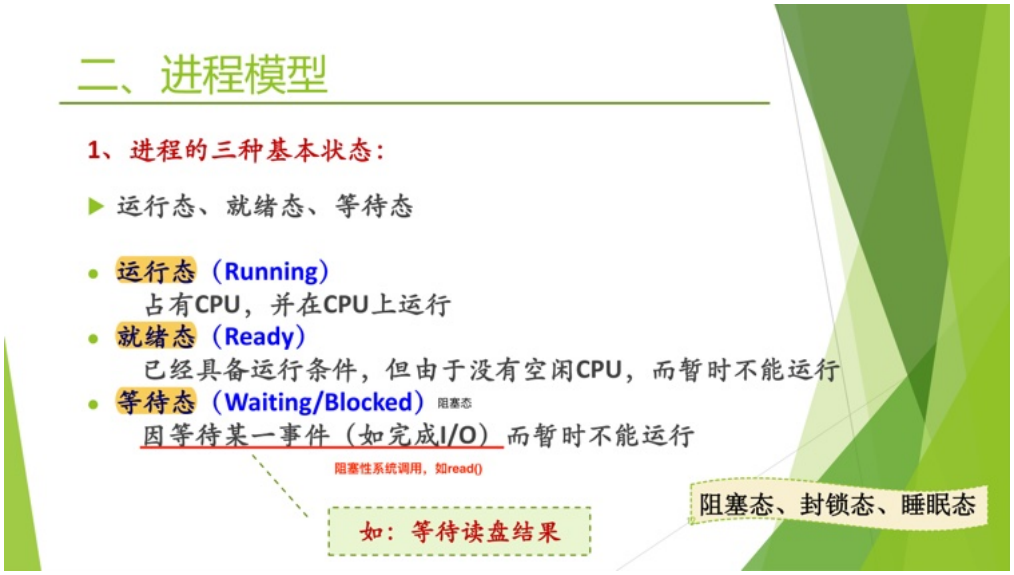
图示：进程的多种分类，以及进程与程序的区别。

进程与程序的区别：

维度	程序	进程
状态	静态	动态
生命	长期存在的文件	有创建和消亡
表达并发	不能	可以
对应关系	1 个	1 个程序可对应多个进程
创建能力	不能创建程序	进程可创建子进程

3.4 进程状态及其转换

3.4.1 三种基本状态



图示：运行 / 就绪 / 等待三态及其语义。

- **运行态 Running**：占有 CPU 并正在执行；

- **就绪态 Ready**：已具备运行条件但暂未分到 CPU；
- **等待 / 阻塞态 Waiting / Blocked**：因等待事件而暂停执行（也称阻塞、封锁、睡眠态）。

阻塞通常由“**阻塞性系统调用**”引发，例如读磁盘、`wait()`、`accept()` 等。

3.4.2 三态之间的转换



图示：转换边及触发条件——调度、抢占、阻塞、唤醒。

转换规则：

- `Ready -> Running`：被调度器选中。
- `Running -> Ready`：时间片到期、出现更高优先级进程、显式让出。
- `Running -> Waiting`：执行了阻塞性系统调用，例如 `read`、`wait`、`acquire lock`。
- `Waiting -> Ready`：等待事件发生，例如 I/O 完成、信号量释放、子进程退出。

注意几个常被搞错的点：

- **阻塞态不能直接到 CPU 运行**，必须先变成就绪。
- **就绪不会自己变成阻塞**：它在排队中，没有“主动等待”的语义。
- 任何 `Running -> Waiting` 都伴随着另一个进程 `Ready -> Running`。

3.4.3 创建态、终止态、挂起态

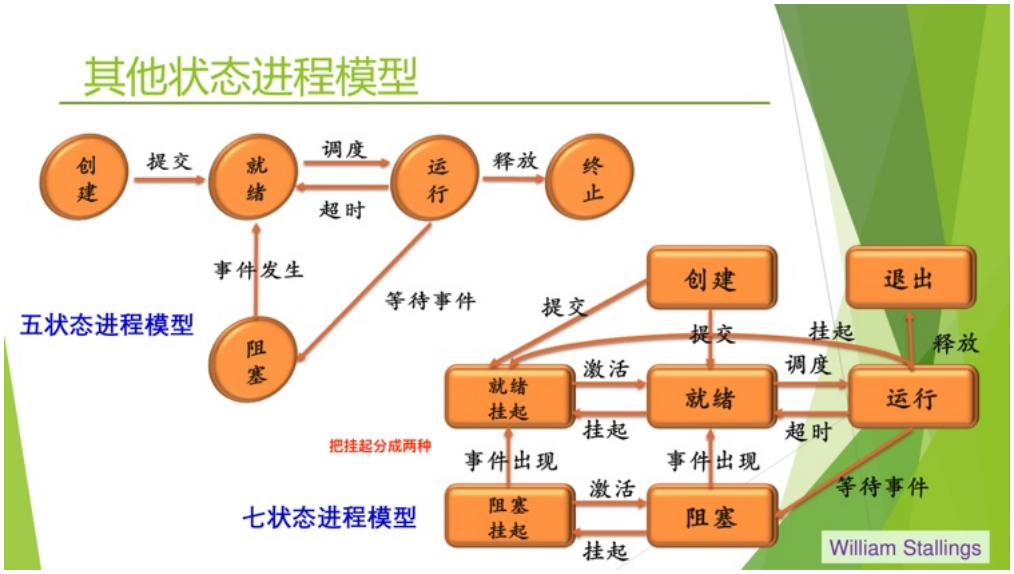
- **创建态 New**：已分配 PID 和 PCB，但尚未被同意执行（资源未到位）。
- **终止态 Terminated**：进程结束执行，进入状态收尾，等待资源回收。
- **挂起态 Suspended**：进程映像被换出到磁盘，不占用内存。挂起可以再细分：
  - 挂起就绪（Suspended Ready）；
  - 挂起阻塞（Suspended Blocked）。



图示：创建态、终止态、挂起态的含义。

挂起的目的：调节系统负载，缓解内存压力。

3.4.4 五状态与七状态模型

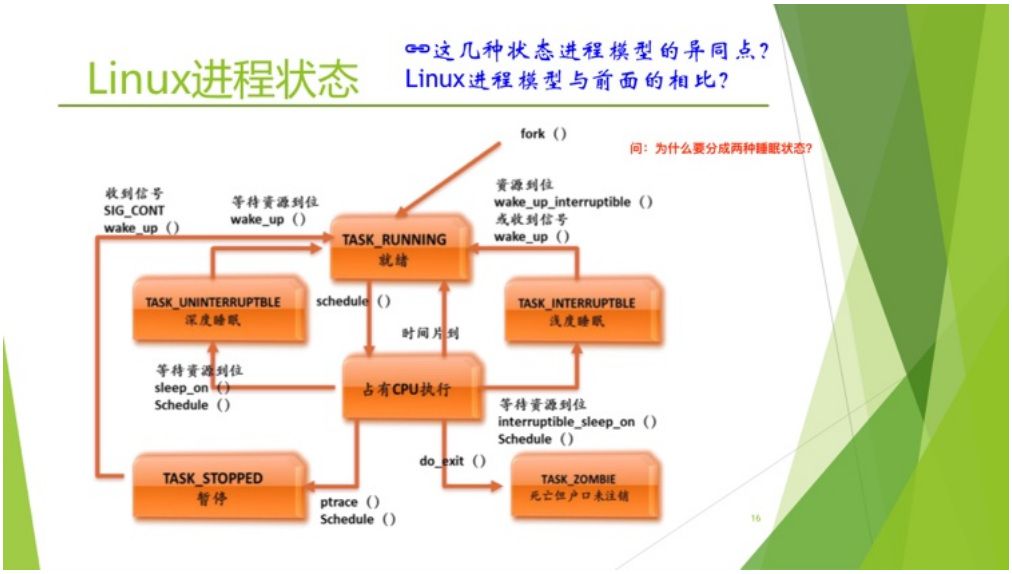


图示：W. Stallings 给出的两个扩展模型。

- **五状态模型**：在三态基础上加 `New` 和 `Terminated`。
- **七状态模型**：再把挂起态拆为 `Suspended Ready` 和 `Suspended Blocked`，多出两条边：
  - `Ready -> Suspended Ready`；
  - `Waiting -> Suspended Blocked`，事件发生后变 `Suspended Ready`，再 `Activate` 回到 `Ready`。

> **理解要点：**多出来的状态都对应“**操作系统在内存上做选择**”这个新策略，状态转换图本身依旧是机制。

3.4.5 Linux 进程状态



图示：Linux 中 R / S / D / T / Z 等状态。

- R (TASK_RUNNING)：运行 / 就绪。
- S (TASK_INTERRUPTIBLE)：可中断睡眠（可被信号唤醒）。
- D (TASK_UNINTERRUPTIBLE)：不可中断睡眠（多用于 I/O）。
- T (TASK_STOPPED / TRACED)：被信号或调试器停止。
- Z (TASK_DEAD - ZOMBIE)：已退出但未被父进程回收。

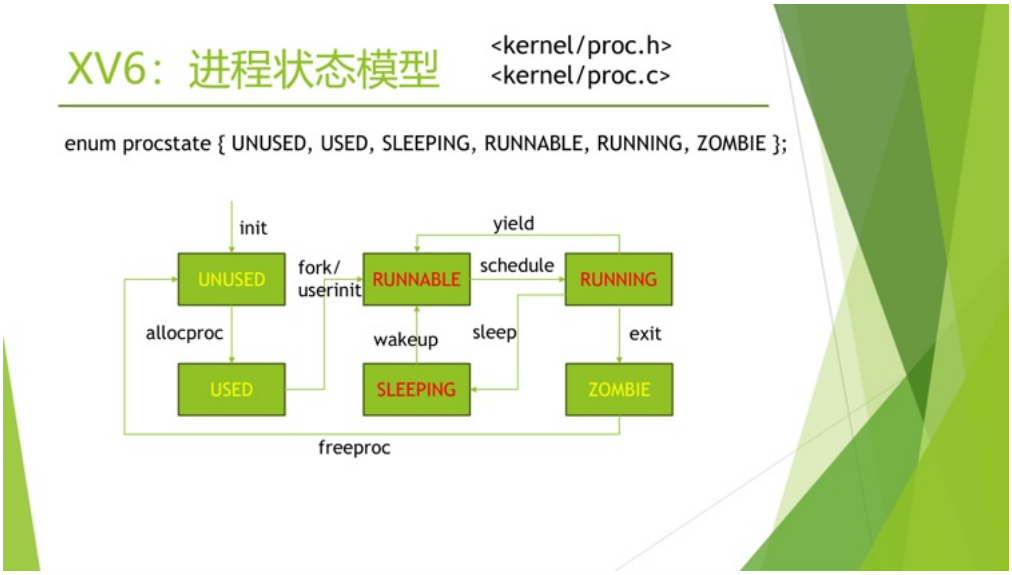
> **思考：**为什么要分两种睡眠？因为某些 I/O 必须保证“原子性”，不希望被信号打断（如磁盘写）；而大多数等待是可被信号唤醒的。

3.4.6 XV6 进程状态

```
enum procstate {
    UNUSED,
    USED,
    SLEEPING,
    RUNNABLE,
    RUNNING,
    ZOMBIE,
};
```

转换：

- UNUSED -> USED：allocproc 分配 proc 槽；
- USED -> RUNNABLE：fork / userinit 完成进程初始化后置为可运行；
- RUNNABLE -> RUNNING：调度器从就绪队列中选中并切换上下文；



图示：XV6 中的 procstate 枚举与状态转换。

- `RUNNING -> SLEEPING`：调用 `sleep`；
- `SLEEPING -> RUNNABLE`：`wakeup` 唤醒；
- `RUNNING -> RUNNABLE`：`yield`（被时钟中断触发）；
- `RUNNING -> ZOMBIE`：`exit` 后等待父进程回收；
- `ZOMBIE -> UNUSED`：`freeproc` 回收槽位。

3.5 进程控制块 PCB

3.5.1 PCB 概念



图示：PCB（task_struct / EPROCESS / proc_t）是 OS 感知进程存在的唯一标志。

- **PCB（Process Control Block）**：又称进程描述符或进程属性，是 OS 中表示进程的核心数据结构。

- PCB 与进程一一对应；操作系统通过它**控制和管理**进程。
- 所有 PCB 的集合构成**进程表**。
- 不同 OS 的具体名字：Linux `task_struct`、Windows `EPROCESS / KPROCESS / PEB`、Solaris `proc_t`。

3.5.2 PCB 的主要内容



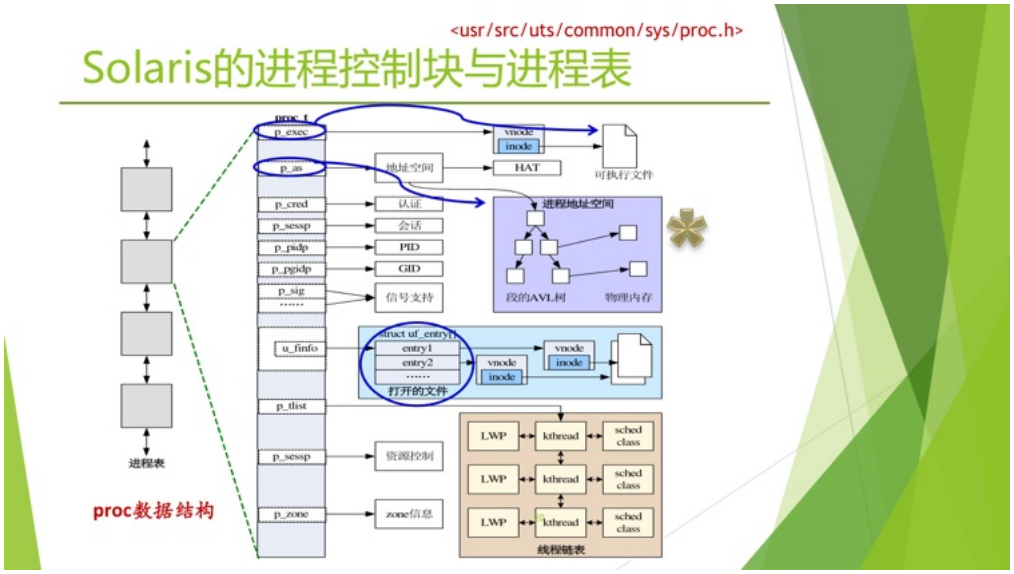
图示：PCB 包含描述、控制、资源、通信、CPU 现场等多类字段。

PCB 通常包含：

- **进程描述信息**：
  - PID（唯一）；
  - 进程名（基于可执行文件名，不唯一）；
  - 用户标识符 UID、进程组关系。
- **进程控制信息**：
  - 当前状态、优先级（基本优先级 + 当前优先级，可被提升）；
  - 代码执行入口地址；
  - 可执行文件名（磁盘地址）；
  - 运行统计信息（执行时间、缺页次数等）。
- **进程间同步与通信**：
  - 阻塞原因；
  - 进程的队列指针；
  - 消息队列指针。
- **所拥有的资源和使用情况**：
  - 虚拟地址空间现状；
  - 打开文件列表；

  - 占用的设备列表。
- **CPU 现场信息（context）**：
  - 通用寄存器、PC、PSW、栈指针；
  - 段 / 页表指针；
  - 进程暂时不运行时由 OS 保存。

3.5.3 Solaris 与 Linux 的具体 PCB



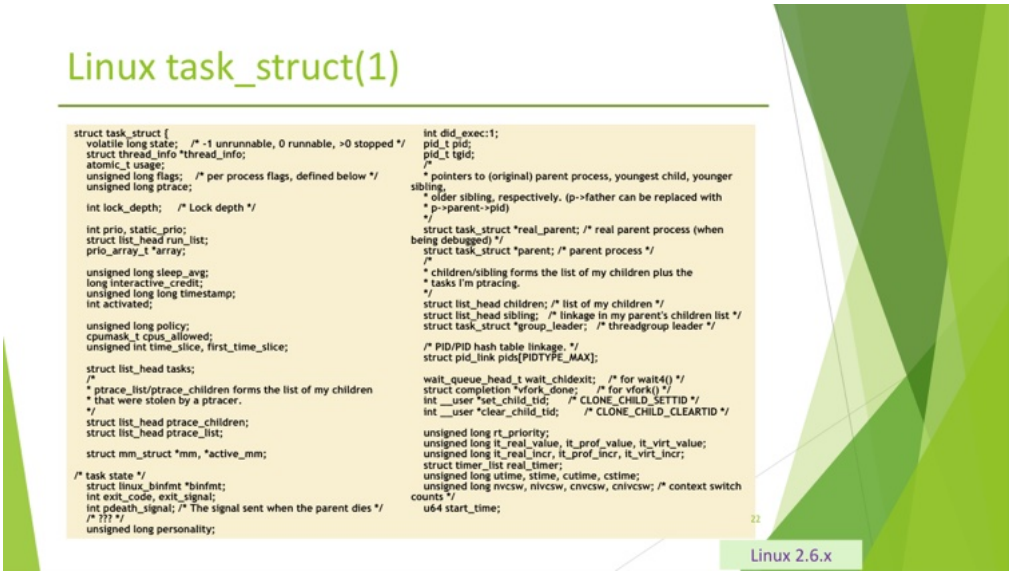
图示：`proc_t` 结构与进程地址空间、文件表、线程链表的关系。

Solaris 的 `proc_t`（位于 `usr/src/uts/common/sys/proc.h`）连接：

- 地址空间（`p_as`）-> 段的 AVL 树 -> 物理内存；
- 凭证（`p_cred`）；
- 会话与 PID（`p_sessp`、`p_pidp`、`p_pgidp`）；
- 信号（`p_sig`）；
- LWP / kthread / sched class 链表（`p_tlist`）；
- 打开文件表（`u_finfo` -> `uf_entry[]` -> vnode/inode）；
- zone 信息、资源控制等。

`task_struct` 的几类典型字段：

- 状态：`volatile long state`；
- 调度：`prio, static_prio, run_list, sleep_avg`；
- 内存：`mm`（指向 `mm_struct`）、缺页统计 `min_flt, maj_flt`；
- 凭证：`uid, euid, gid, egid, cap_*`；
- 信号、文件、命名空间、cgroups 等。



图示：`task_struct` 字段示例，从状态到内存到信号、credential 等。

3.5.4 XV6 PCB



图示：XV6 中 `struct proc` 字段以及与真实操作系统的对比。

XV6 的 `struct proc`（`kernel/proc.h`）核心字段：

- `state`：上面的 `procstate`；
- `pid`：进程 ID；
- `parent`：父进程；
- `pagetable`：页表；
- `kstack`：内核栈；
- `trapframe`：中断/异常时保存的用户寄存器；
- `context`：内核上下文（用于内核切换）；
- `chan`：sleeping 时等待的对象指针；

- `killed`：是否被请求退出；
- `xstate`：退出状态；
- `ofile[]`、`cwd`：打开文件、当前目录；
- `name[16]`：进程名。

> 真实场景的 PCB 远比 XV6 复杂——会包含调度优先级、命名空间、cgroup、信号处理、安全凭证等。

3.6 进程地址空间

3.6.1 概念与示例



图示：每个进程都看到一个独立的虚拟地址空间。

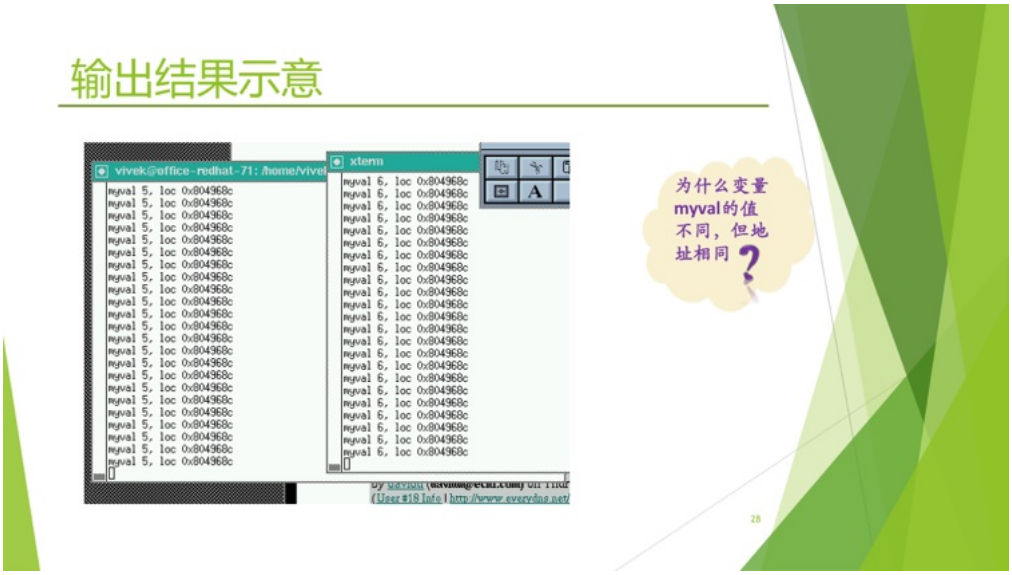
观察一段简单程序：

```c
int myval;
int main(int argc, char *argv[]) {
    myval = atoi(argv[1]);
    while (1)
        printf("myval is %d, loc 0x%lx\n", myval, (long)&myval);
}
```

同时运行两次：

```
./myval 5
./myval 6
```

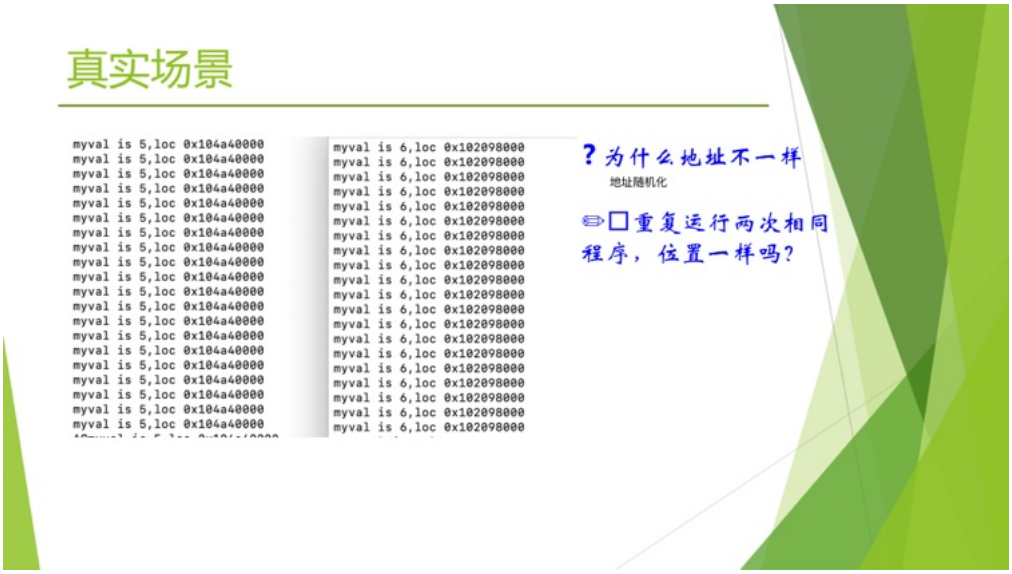
- 两次进程看到的 `&myval` 在打印上常常相同；
- 但它们对应的物理页是不同的——这是“**虚拟地址相同但物理位置独立**”的直接证据；



图示：两个进程打印的 myval 值不同，但 `&myval` 看起来是同一个。

- 即“地址空间是独立的”，是进程隔离的基础。

3.6.2 ASLR 与重复运行的影响

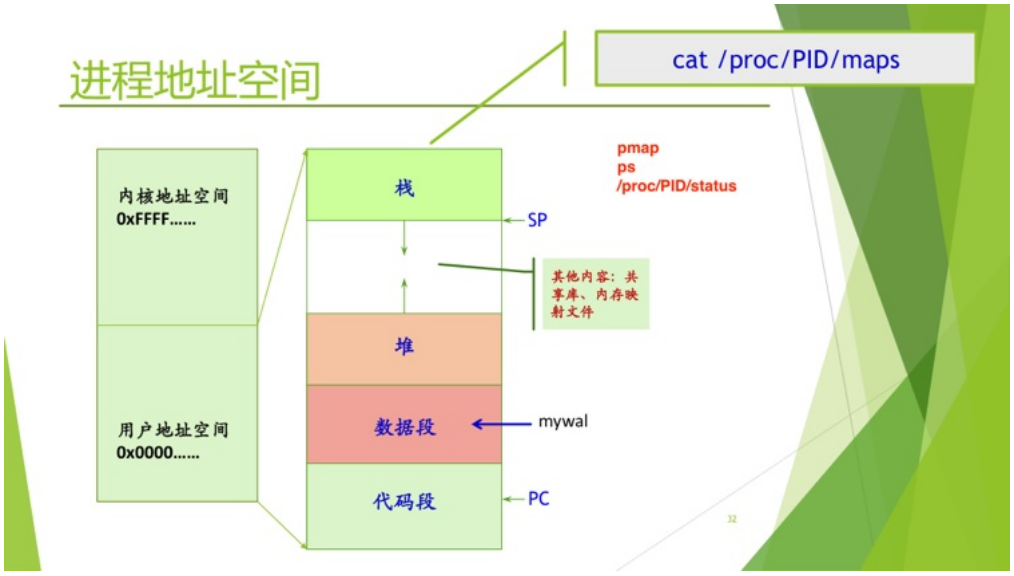


图示：ASLR 让多次运行同一程序看到的地址不同。

- **ASLR（Address Space Layout Randomization）**是地址空间随机化机制；
- 同一程序多次启动会看到不同地址，提升对漏洞利用的防护；
- 实验室验证可临时关闭：

```
sudo sysctl -w kernel.randomize_va_space=0
```

3.6.3 进程地址空间的典型布局



图示：从低到高依次是代码、数据、堆、共享库、栈，最高地址为内核空间。

```
高地址 +--------------------------+
       |      内核地址空间          |
       +--------------------------+
       |      栈（向下生长）         |
       |           v              |
       | -- 共享库 / mmap 区域 --   |
       |           ^              |
       |      堆（向上生长）         |
       +--------------------------+
       |     已初始化数据 .data      |
       |     未初始化数据 .bss       |
       +--------------------------+
       |     代码段 .text           |
低地址 +--------------------------+
```

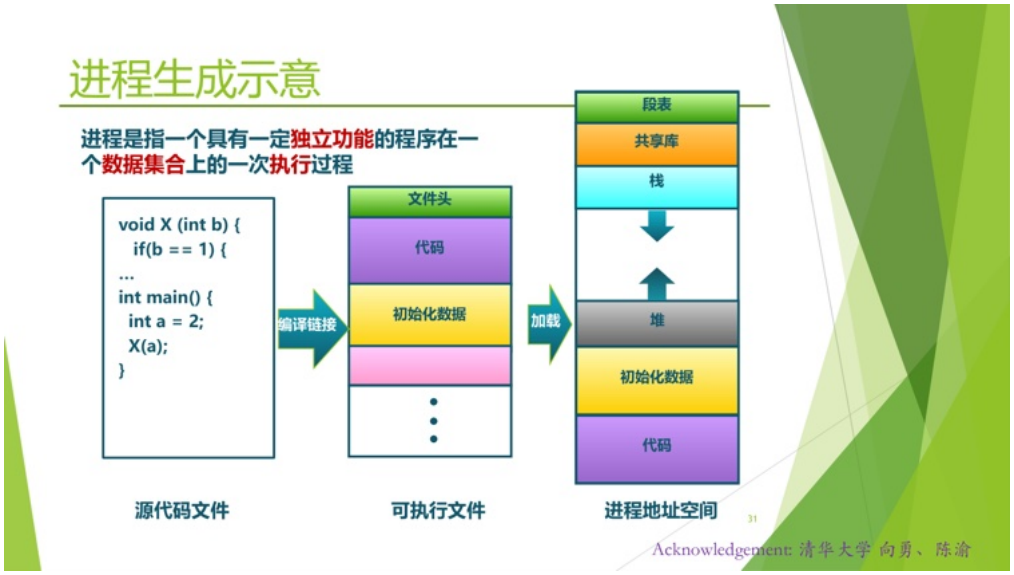
可执行文件经过编译链接得到段表，加载时映射到上述布局。

3.6.4 /proc/<pid>/maps 与 VMA

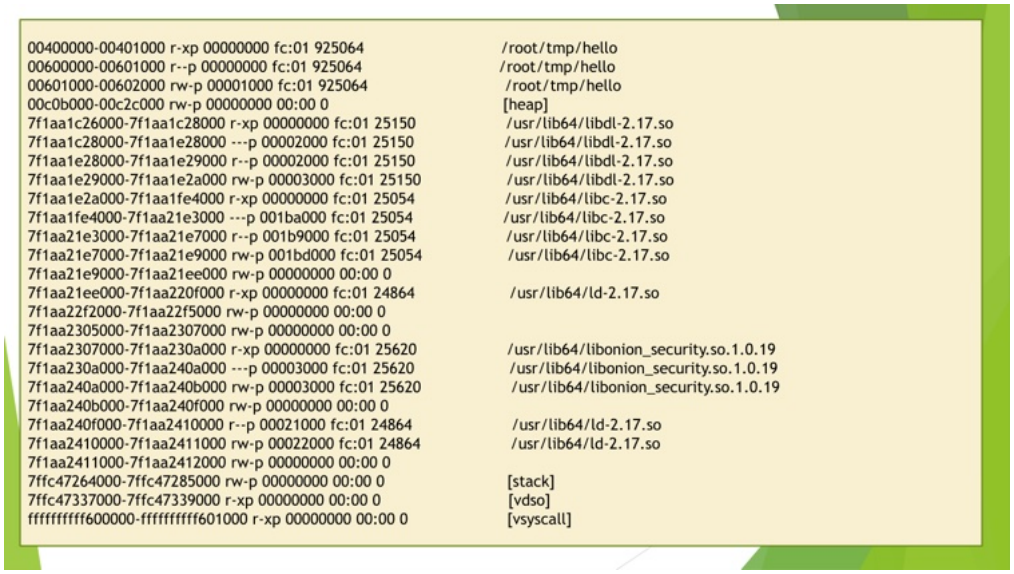
```
00400000-00401000 r-xp 00000000 fc:01 925064 /root/tmp/hello
00600000-00601000 r--p 00000000 fc:01 925064 /root/tmp/hello
00601000-00602000 rw-p 00001000 fc:01 925064 /root/tmp/hello
00c0b000-00c2c000 rw-p 00000000 00:00 0      [heap]
7f1...                                       [stack]
```

每行表示一个 VMA，包含：起止地址、权限、偏移、设备、inode、映射文件名。

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
```



图示：源代码经过编译链接得到带段表的可执行文件，加载到进程地址空间。

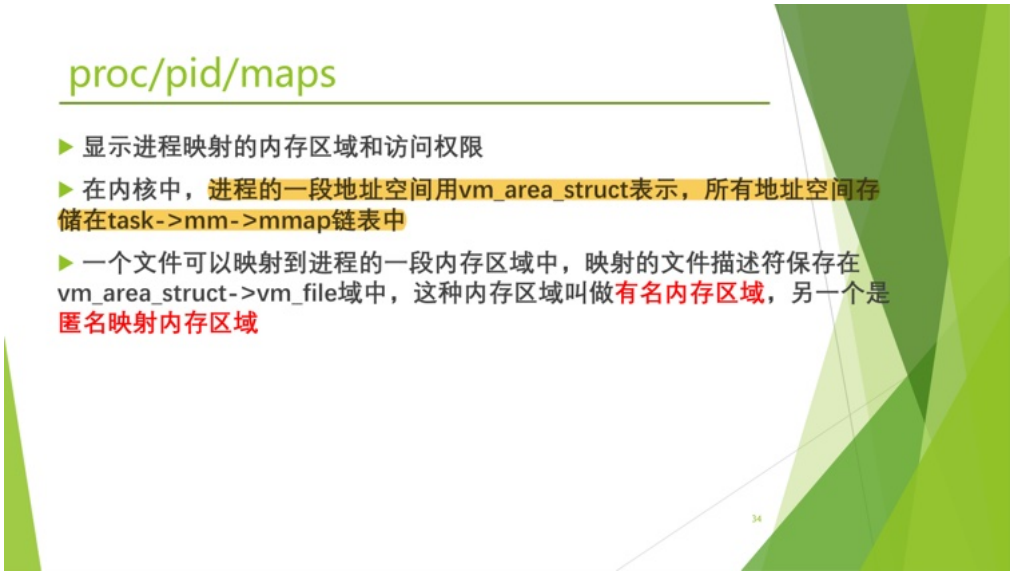


图示：通过 /proc/<pid>/maps 可以看到进程实际的虚拟内存区域、权限和映射文件。

```
    printf("hello, world!");
    int *a = malloc(sizeof(int));
    getchar();    // 阻塞，便于观察 /proc/<pid>/maps
    return 0;
}
```

这段程序同时使用了：

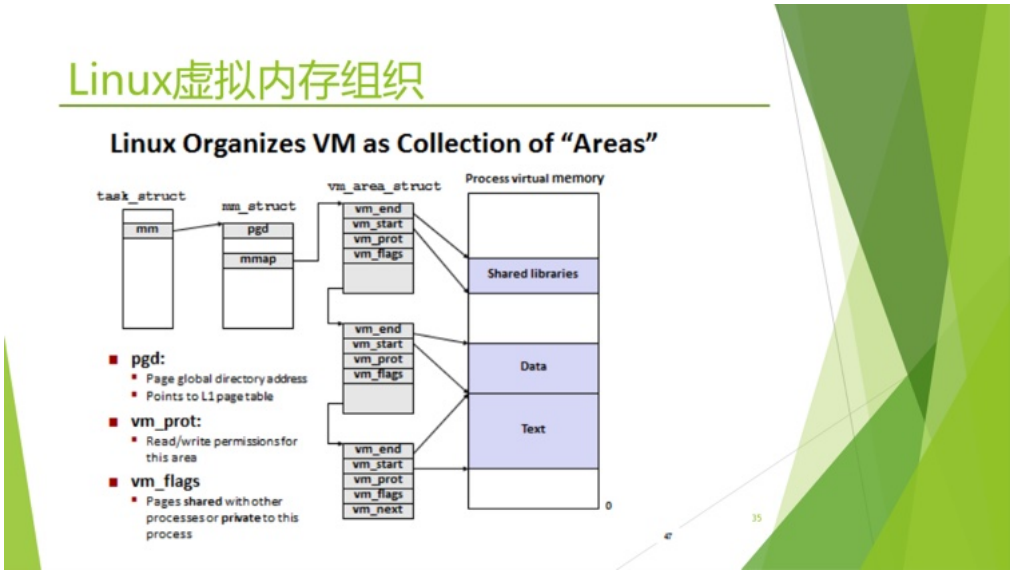
- 常量字符串（.rodata / .text 区）；
- 堆（malloc）；
- 阻塞调用（getchar）让进程不会立即退出。



图示：内核通过 `vm_area_struct` 描述地址空间，每段挂在 `task->mm->mmap` 链表上。

3.6.5 Linux 内核层面的地址空间

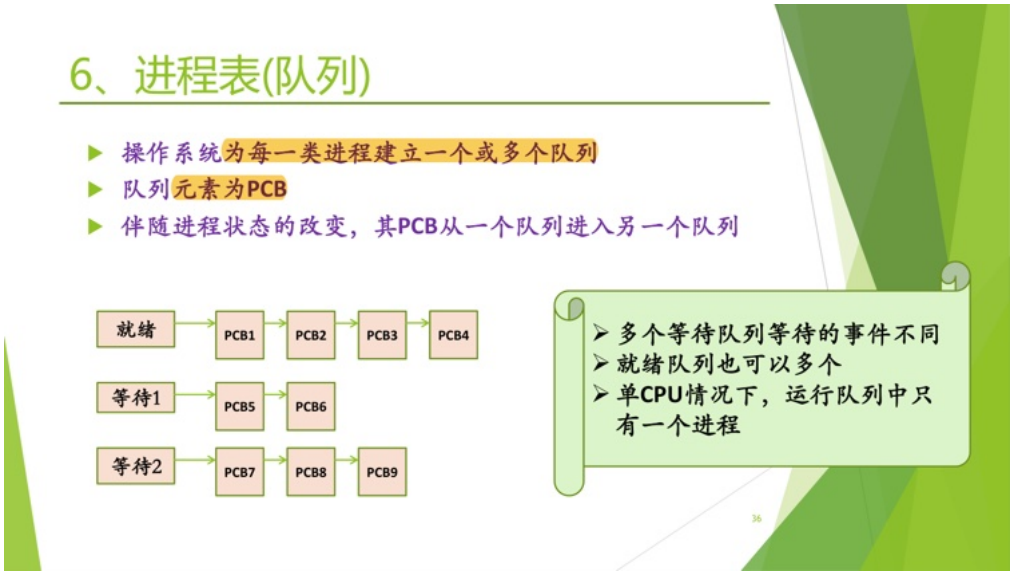
- 在内核中，进程的一段地址空间用 `vm_area_struct` 表示。
- 所有 VMA 挂在 `task->mm->mmap` 链表（或 maple tree）上。
- 一个文件可以映射到进程的一段内存区域，映射的文件描述符保存在 `vm_area_struct->vm_file` 中——**有名内存区域**；没有 `vm_file` 的是匿名内存区域。



图示：用户进程地址空间由 `mm_struct` + `vm_area_struct` 描述，再通过页表与物理页对应。

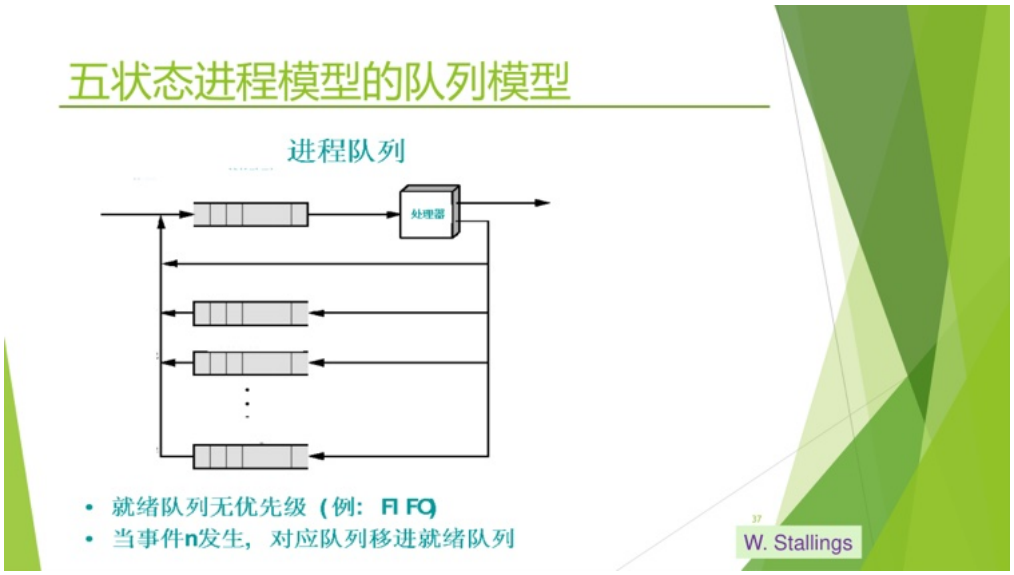
3.7 进程队列模型

- 操作系统按状态把进程分进若干队列：
  - **就绪队列**：可能多个，按优先级或调度类划分；
  - **等待队列**：通常按等待事件分类，每类事件一个队列；



图示：每类状态的进程通常组织成队列，PCB 在队列间迁移。

  - **运行队列**：单 CPU 时只放当前正在运行的进程。
- 状态变化对应队列迁移。



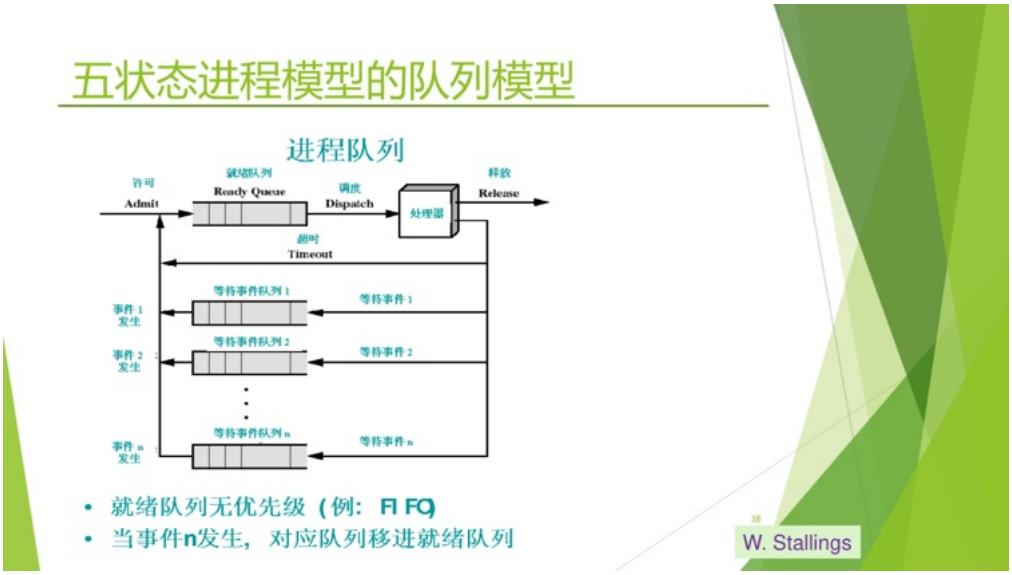
图示：五状态模型对应的队列结构。

> **要点：**队列模型把状态机变成具体的数据结构，是调度的实现基础。

3.8 进程控制

3.8.1 原语

- 进程控制操作完成进程状态间的转换，由具有特定功能的**原语**完成。
- **原语（Primitive）**：完成某种特定功能的一段程序，**不可分割 / 不可中断**，即原子操作（Atomic）。



图示：阻塞队列与就绪队列的协作。



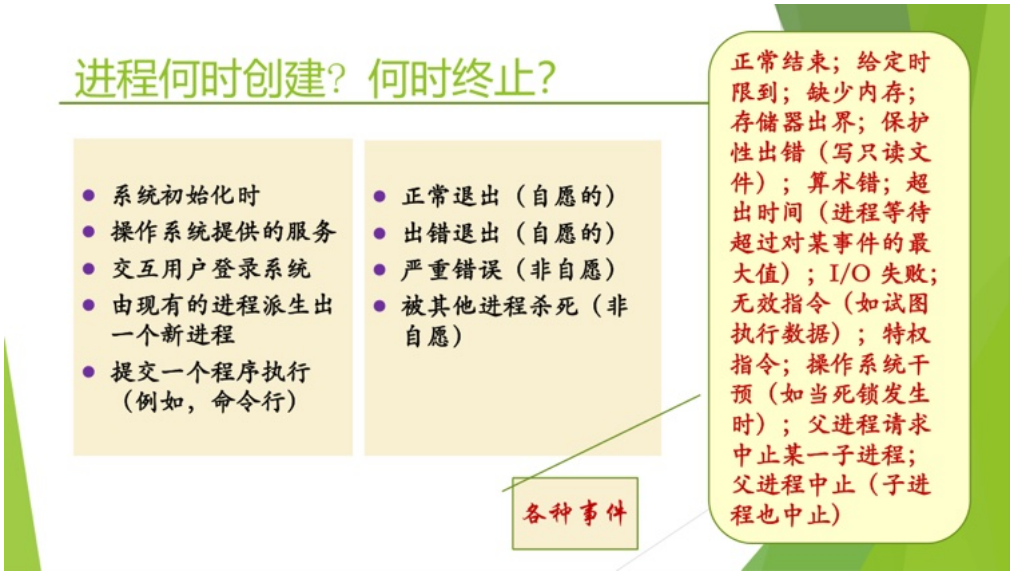
图示：原语是不可中断、原子完成的内核操作。

- 典型原语：进程创建、进程撤消、阻塞、唤醒、挂起、激活、改变优先级。
- 原语执行期间不响应一般中断，避免破坏内核数据结构的一致性。

3.8.2 何时创建 / 何时终止

创建时机：

- 系统初始化时；
- 操作系统提供的服务；
- 用户登录系统；
- 现有进程派生出新进程；
- 提交一个程序执行（如命令行）。



图示：典型创建场景与终止场景。

终止时机：

- 正常退出（自愿）；
- 出错退出（自愿）；
- 严重错误（非自愿）；
- 被其他进程杀死（非自愿）。

可触发终止的事件包括：正常结束、给定时限到、缺少内存、存储器越界、保护性出错、算术错、超时、I/O 失败、无效指令、特权指令、OS 干预（如死锁）、父进程请求中止子进程、父进程中止。

3.8.3 进程创建原语



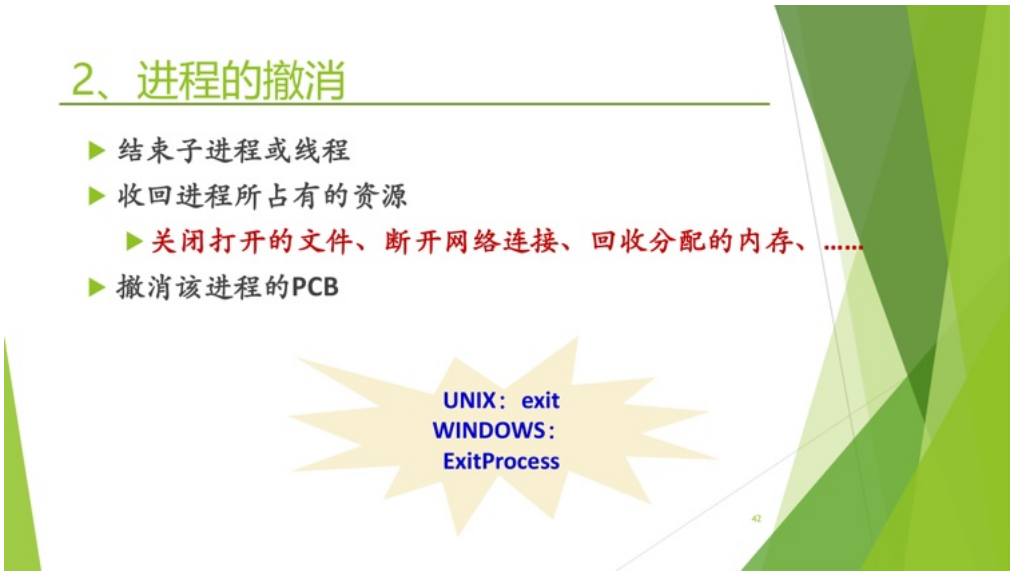
图示：UNIX 是 fork+exec，Windows 是 CreateProcess。

进程创建主要工作：

1. 给新进程分配唯一 PID 与 PCB；
2. 为进程分配地址空间；
3. 初始化 PCB（默认状态、优先级、队列指针等）；
4. 把新进程加入就绪队列；
5. 创建或扩充其他数据结构（打开文件、信号、命名空间等）。

UNIX：`fork()` + `exec()`；Windows：`CreateProcess()`。

3.8.4 进程撤消原语



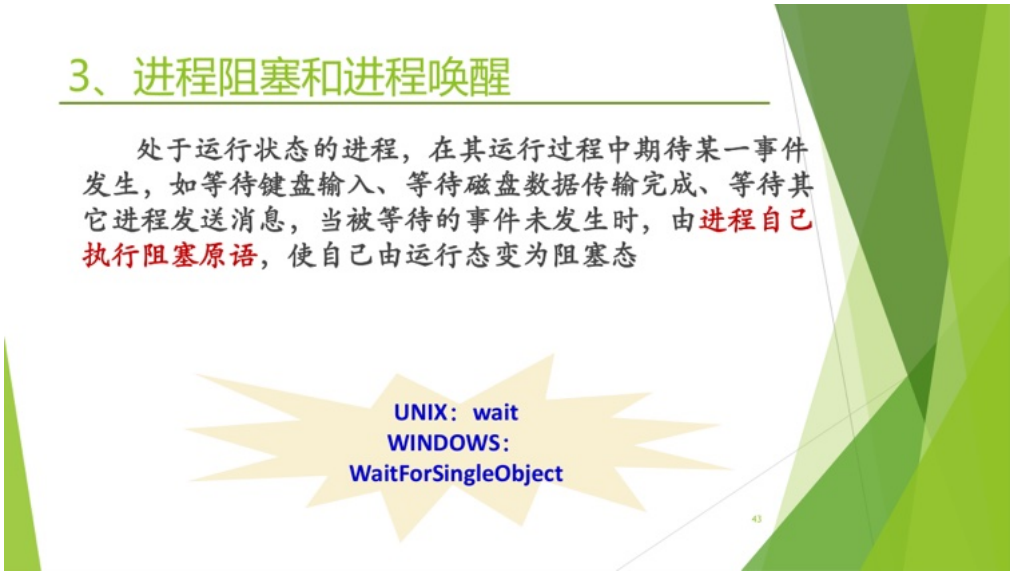
图示：UNIX `exit`，Windows `ExitProcess`。

撤消进程要做的工作：

1. 结束子进程或线程；
2. 收回所占资源（关闭打开文件、断开网络、释放内存）；
3. 撤消该进程的 PCB；
4. 释放它在就绪 / 等待队列上的位置；
5. 通知父进程（产生 `SIGCHLD`，或让父进程的 `wait` 返回）。

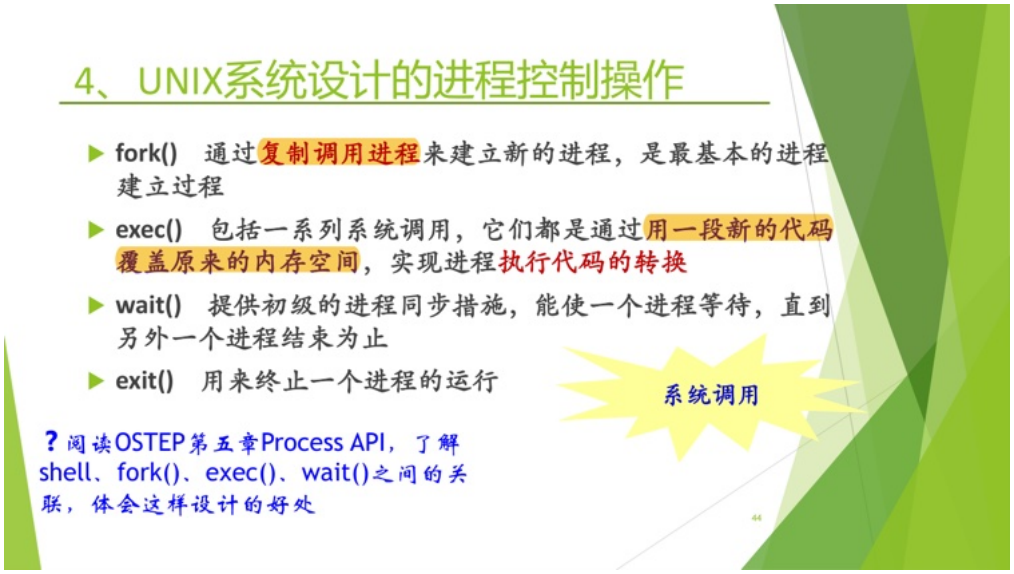
3.8.5 进程阻塞与唤醒

- 阻塞由**进程主动**触发：执行阻塞性系统调用，例如 `wait()`、`read()`、`acquire(lock)`。
- 唤醒由**他者**触发：另一个进程发消息、释放锁，或者中断处理程序在 I/O 完成后调用唤醒原语。
- UNIX：`wait()`；Windows：`WaitForSingleObject()`。



图示：阻塞由进程自己执行；唤醒通常由其他进程或中断处理程序执行。

3.8.6 UNIX 系统调用：fork / exec / wait / exit



图示：fork/exec/wait/exit 之间的协作关系。

- `fork()`：复制当前进程（父）得到新进程（子），父子共享代码和初始数据，但拥有独立 PID。
- `exec()`：用一段新的代码覆盖原内存空间，使该进程开始执行新程序。
- `wait()`：父进程等待某个子进程结束，回收其资源；提供初级同步。
- `exit()`：终止当前进程，释放资源。

> **设计哲学：“fork + exec 分离”** 是 UNIX 的关键设计。`fork()` 后子进程可以先调整文件描述符、信号设置等，再调用 `exec()` 加载新程序，使得 shell 实现重定向、管道等极为简洁。

OSTEP 第 5 章问题

> **代码阅读题：**阅读 OSTEP 第 5 章 *Process API* 与 XV6 `kernel/proc.c`、`user/sh.c`，说明 shell、`fork()`、`exec()`、`wait()` 之间的关联，并体会这种设计的好处。

解答：

- shell 通过解析命令行得到要执行的程序与参数。
- 对每条命令，shell 大致执行：
  1. `pid = fork()` 复制自身，得到子进程；
  2. 子进程内：
     - 若需要重定向，先 `close(fd)` 再 `open` 或 `dup` 文件描述符；
     - 若需要管道，先 `pipe()` 后 `dup2`；
     - 调用 `exec()` 加载真正要运行的程序；
  3. 父进程（shell）：调用 `wait(&status)` 等待子进程结束并获取退出状态。
- 设计好处：
  - **职责单一**：`fork` 只负责复制环境；`exec` 只负责换程序；`wait` 只负责等待。
  - **灵活组合**：在 `fork` 与 `exec` 之间，shell 可以任意调整子进程环境（重定向、管道、信号、用户切换），不必为每种组合发明新接口。
  - **简洁可组合**：`pipe + fork + dup2 + exec` 即可实现管道命令；`fork + chdir + exec` 即可改变工作目录后启动子程序。
- 在 XV6 `user/sh.c` 中能看到对应的实现：`runcmd` 中根据命令类型派生子进程，并在适当节点调用 `exec` 或 `pipe`；父进程统一在 `runcmd` 末尾或 `main` 中 `wait`。

3.8.7 UNIX `fork()` 实现与 COW 优化

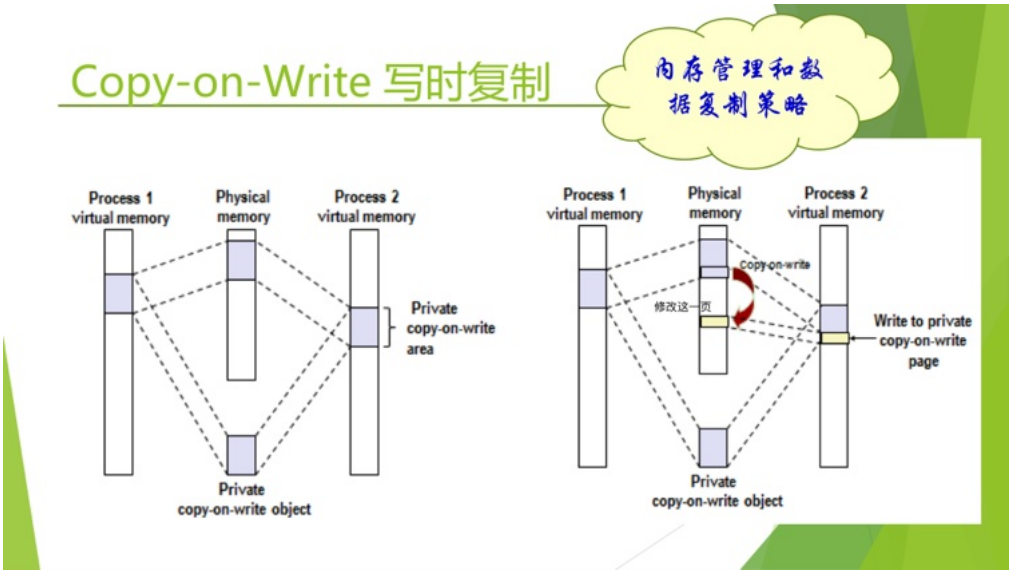


图示：分配 proc、PID，复制地址空间并继承资源，最后入就绪队列。

经典实现：

1. 为子进程分配空闲的 `proc` 结构；
2. 分配唯一 PID；
3. 以一次一页的方式复制父进程地址空间；
4. 继承共享资源（打开文件、当前目录等）；
5. 将子进程状态设为就绪并加入就绪队列；
6. 给子进程返回 0；给父进程返回子进程 PID。

但若紧接着 `exec` 替换程序，整体复制是浪费。Linux 用**写时复制（Copy-on-Write, COW）**优化：



图示：父子进程共享物理页，写时再复制。

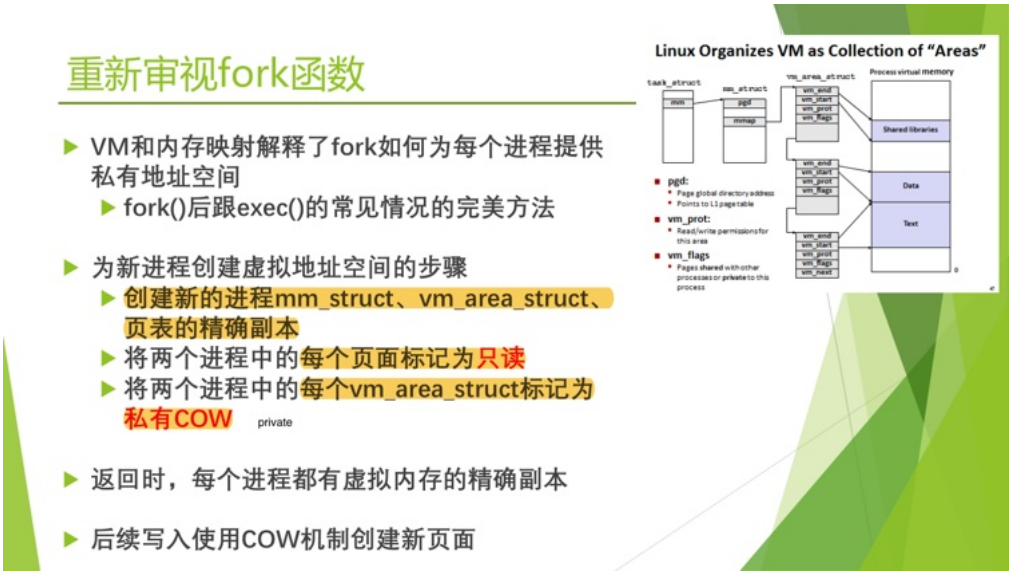
COW 步骤：

1. `fork` 时把父子进程的所有页面权限改成只读，**共享**同一物理页；
2. 当任意一方写入：
   - CPU 检测到写只读页面，触发缺页异常；
   - 内核确认是 COW 页面，分配新物理页框并复制旧内容；
   - 修改对应进程的页表项，把它指向新页面，并恢复可写权限；
   - 退出异常，写操作正常完成；
3. 另一方继续看到原页面。

这一优化对 `fork()` + `exec()` 模型尤其友好——大多数页面在被写之前就被 `exec` 丢弃，从未真正被复制。

虚拟内存视角下 `fork` 的步骤：

1. 创建子进程的 `mm_struct`、`vm_area_struct` 与页表的精确副本；

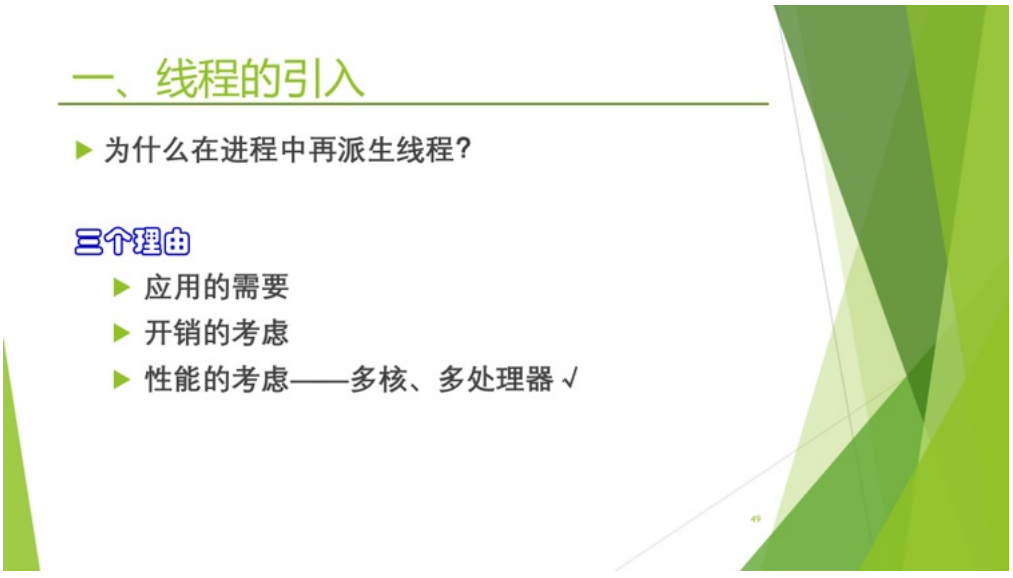


图示：fork 在虚拟内存视角下的完整流程。

2. 把父子进程的每页都标记为只读；
3. 把父子进程每个 VMA 标记为私有 COW；
4. 返回时，每个进程都拥有相同的虚拟内存视图；
5. 后续写入由 COW 创建新页面。

3.9 线程模型

3.9.1 为什么引入线程



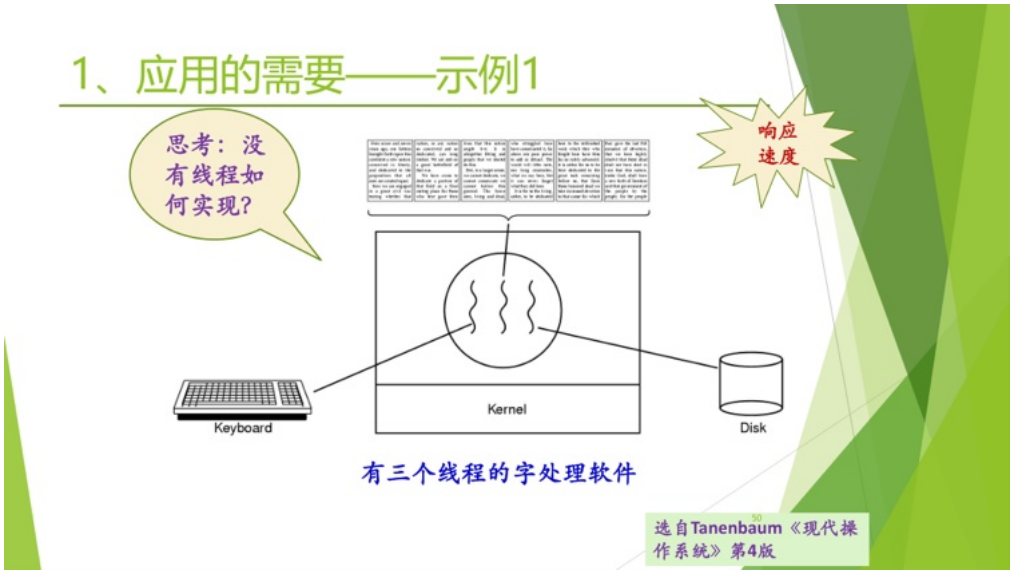
图示：应用需求、开销、性能（多核）。

引入线程主要源于三个动机：

1. **应用需要**：很多程序内部有多个并发任务，例如字处理软件、Web 服务器；
2. **开销考虑**：进程切换太重，线程切换更轻；

3. **性能考虑**：多核时代下需要真正的并行执行。

3.9.2 应用示例：多线程字处理与 Web 服务器

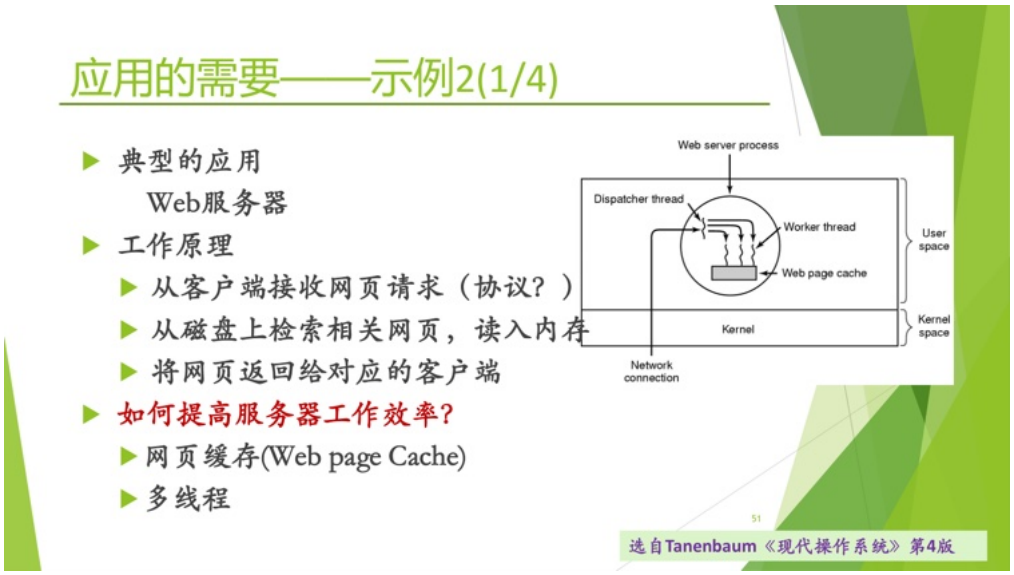


图示：分别处理用户输入、自动保存和拼写检查的三个线程。

字处理软件的三个并发活动：

- 显示线程：实时刷新界面、响应用户输入；
- 后台保存线程：周期性保存文档；
- 拼写检查线程：异步扫描文本。

如果只用一个线程，要么响应慢，要么后台任务被阻塞。



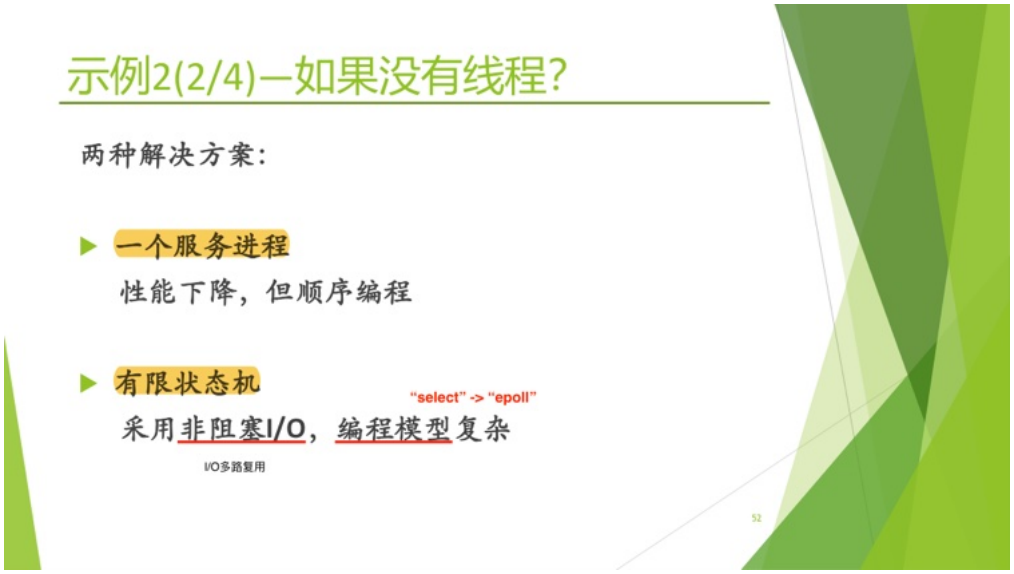
图示：Web 服务器要从客户端接收请求、读文件、回写响应。

Web 服务器的工作流程：

1. 接收来自客户端的请求；

2. 从磁盘检索网页（如缓存命中可直接返回）；
3. 把响应返回客户端。

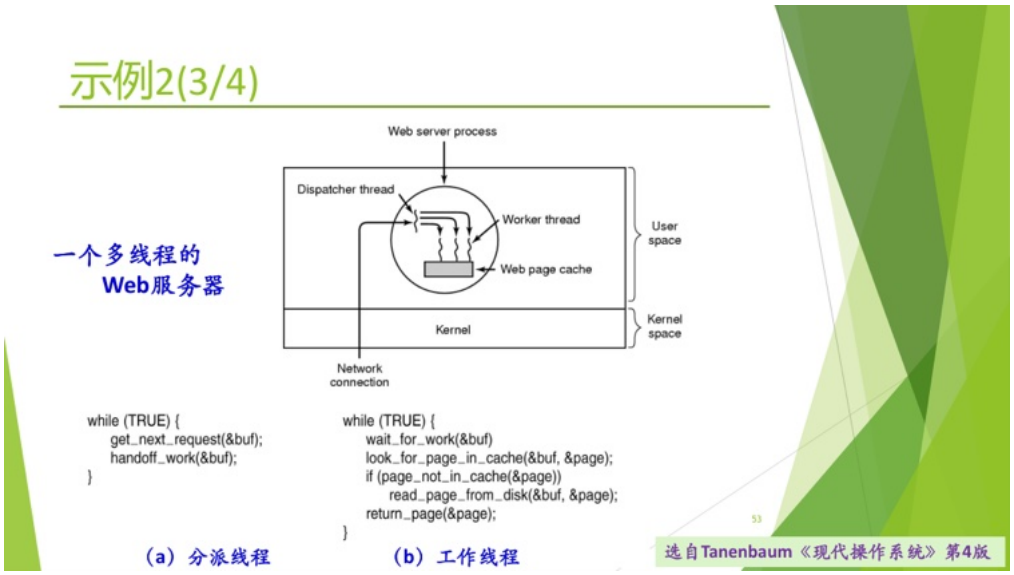
提升效率的两种思路：网页缓存、多线程。



图示：单进程顺序处理 vs. 有限状态机非阻塞 I/O。

无线程的两种解决方案：

- **单服务进程顺序处理**：性能差，但编程模型简单；
- **有限状态机 + 非阻塞 I/O**（如 `select` / `epoll`）：性能好，但编程复杂，是 I/O 多路复用的源头。



图示：分派线程接收请求并交给一组工作线程处理。

多线程 Web 服务器：

- **分派线程（Dispatcher）**：监听端口、接受连接、把请求交给工作线程；

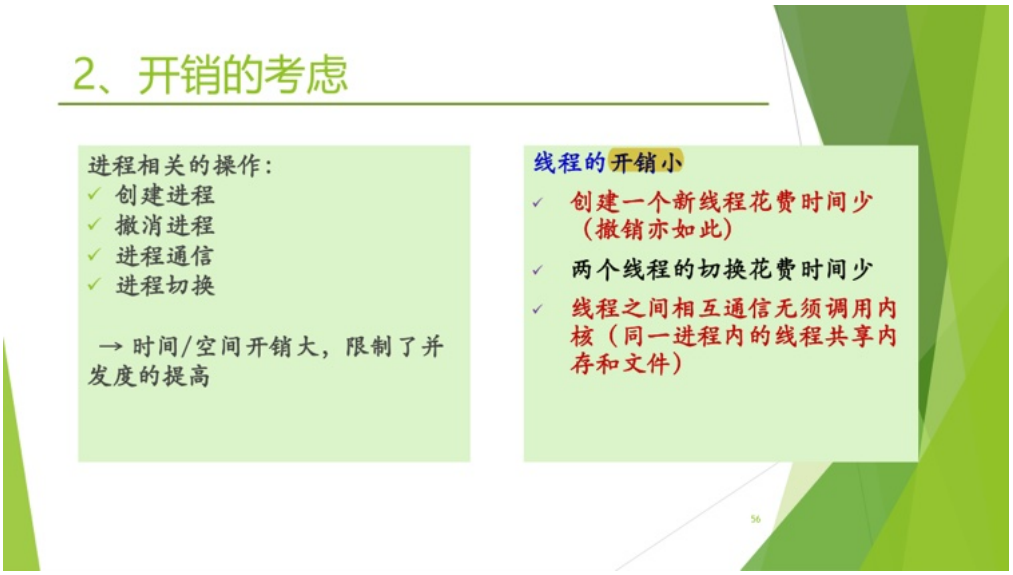
- **工作线程（Worker）**：读文件、构造响应、写回客户端。



图示：多线程 / 单线程进程 / 有限状态机。

模型	并行性	系统调用
多线程	有	阻塞系统调用
单线程进程	无	阻塞系统调用
有限状态机	有	非阻塞系统调用 + 中断

3.9.3 开销的考虑

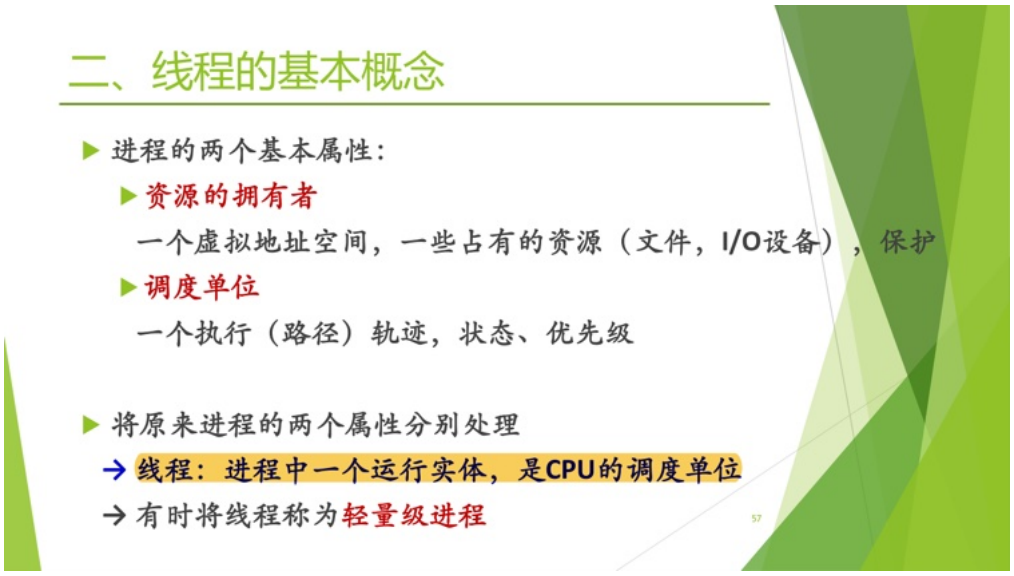


图示：进程创建、撤消、通信、切换的时间和空间开销都较大。

- 进程创建、撤消、切换开销大，限制了并发度；
- 线程：
  - 创建快、销毁快；
  - 切换快（同一地址空间内）；

- 通信快（共享内存即可，无需进入内核）。

3.9.4 线程的基本概念



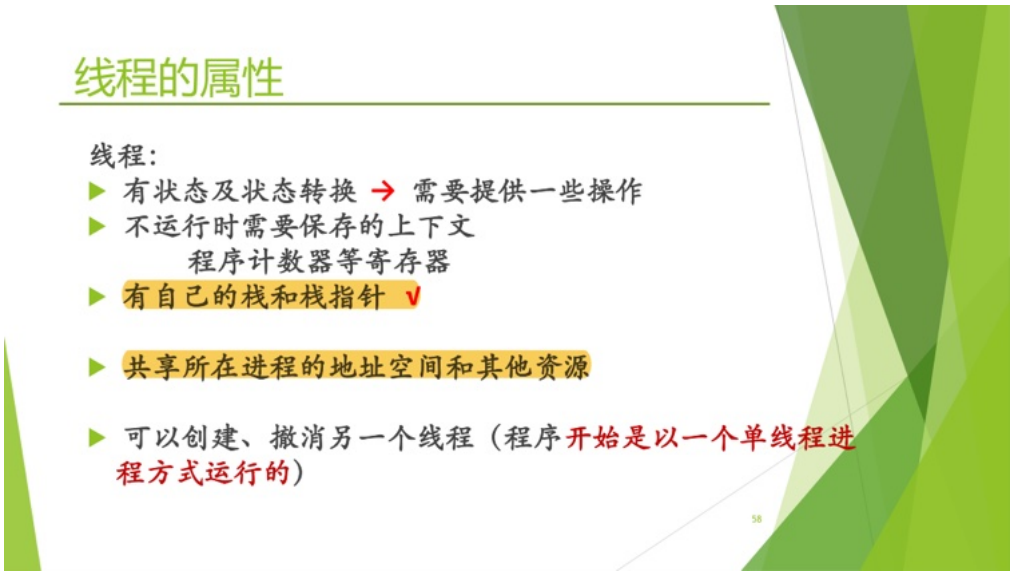
图示：把进程“资源拥有”和“CPU 调度”两属性拆分，得到线程。

进程的两个基本属性：

- **资源拥有者**：地址空间、文件、I/O 设备、保护边界；
- **调度单位**：执行轨迹、状态、优先级。

把这两个属性拆开就得到线程：

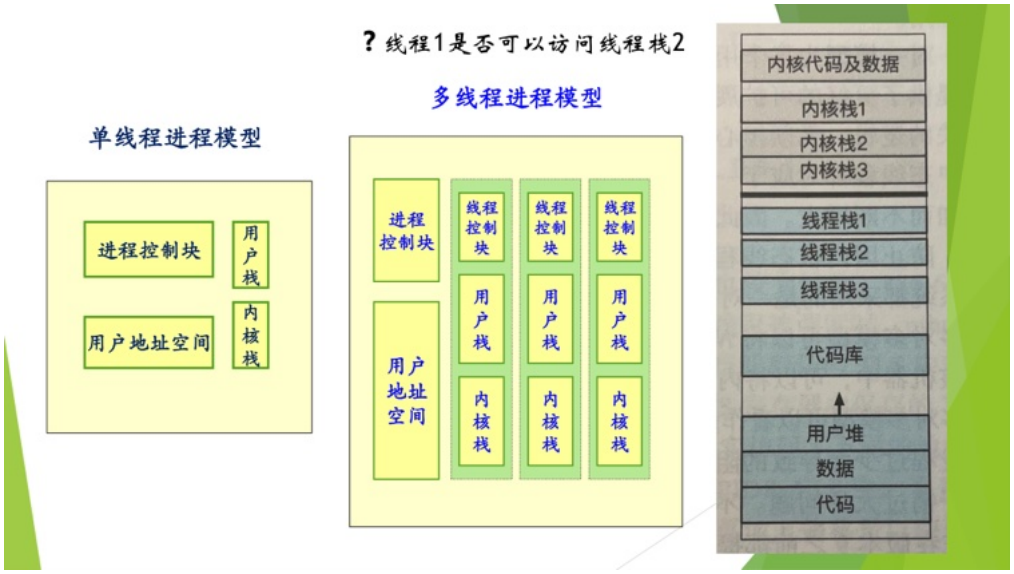
- 进程仍是资源分配单位；
- 线程是 CPU 调度单位，是“轻量级进程（LWP）”。



图示：线程拥有自己的状态、寄存器、栈，但共享进程资源。

线程的属性：

- **有自己的状态及状态转换**；
- **不运行时需要保存的上下文**：PC、寄存器、栈指针、栈本身；
- **共享所在进程的资源**：地址空间、打开文件、信号处理状态等；
- **可以创建、撤消其他线程**；
- 程序通常以单线程方式启动，再创建其他线程。



图示：单线程进程只有一个 PCB；多线程进程拥有进程级 PCB 加上每线程 TCB（线程控制块），每线程有自己的用户栈、内核栈和上下文。

- 单线程进程：一个 PCB + 一份地址空间 + 一份用户栈 + 一份内核栈。
- 多线程进程：一个进程级 PCB + 多个 TCB；每个 TCB 拥有自己的用户栈、内核栈和执行上下文，但共享地址空间。

> **思考：** 线程 1 是否能访问线程 2 的栈?
>
> **解答：** 在多线程进程内，**所有线程共享同一地址空间**，线程 1 拿到线程 2 的栈指针后是能访问的（语言层面通常约定不要这么做）。这正是“多线程方便共享数据，但容易出现竞态”的根本原因。

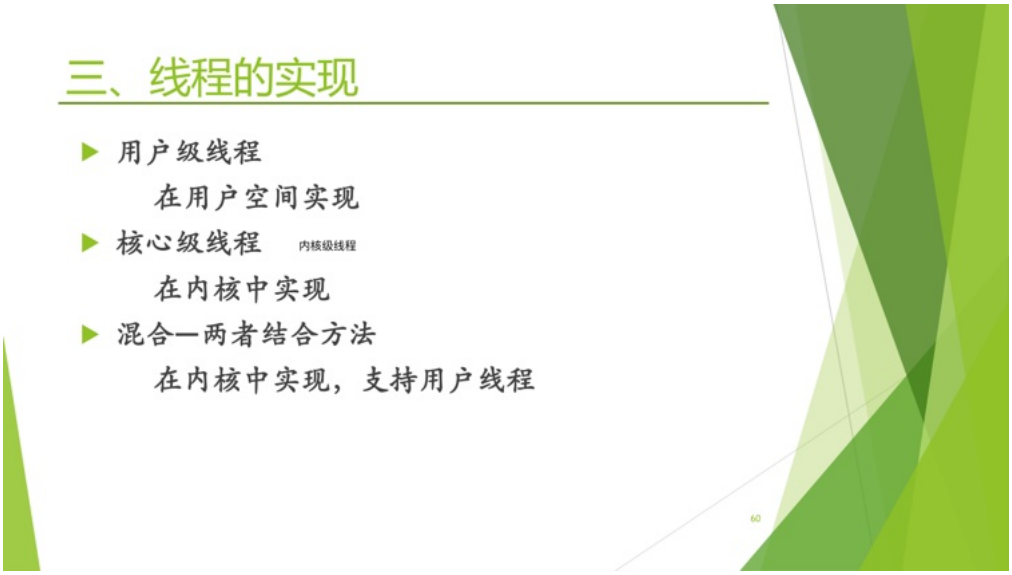
3.9.5 线程实现的三种方式

用户级线程 ULT

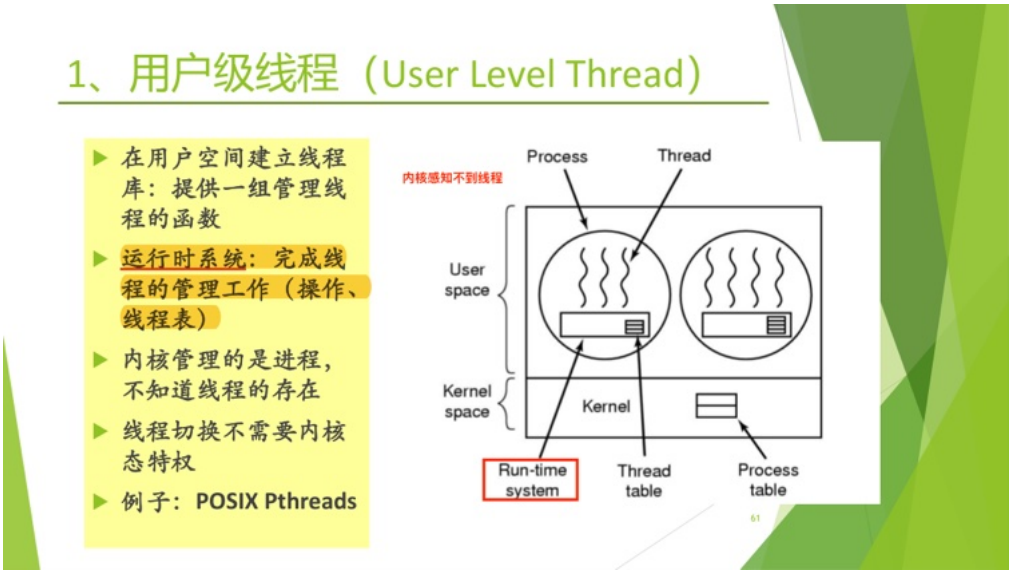
- 在用户空间建立线程库（提供 `create / yield / join / exit` 等）；
- 运行时系统在用户态完成线程调度，维护用户线程表；
- 内核只看到进程；线程切换不需要内核参与，**速度极快**；
- 例子：早期 POSIX Pthreads 的 user-thread 实现、绿色线程（Green Threads）。

典型场景：

- case 1：进程 B 正常运行；



图示：用户级线程、核心级线程、混合实现。



图示：内核只感知进程，线程库在用户空间实现。

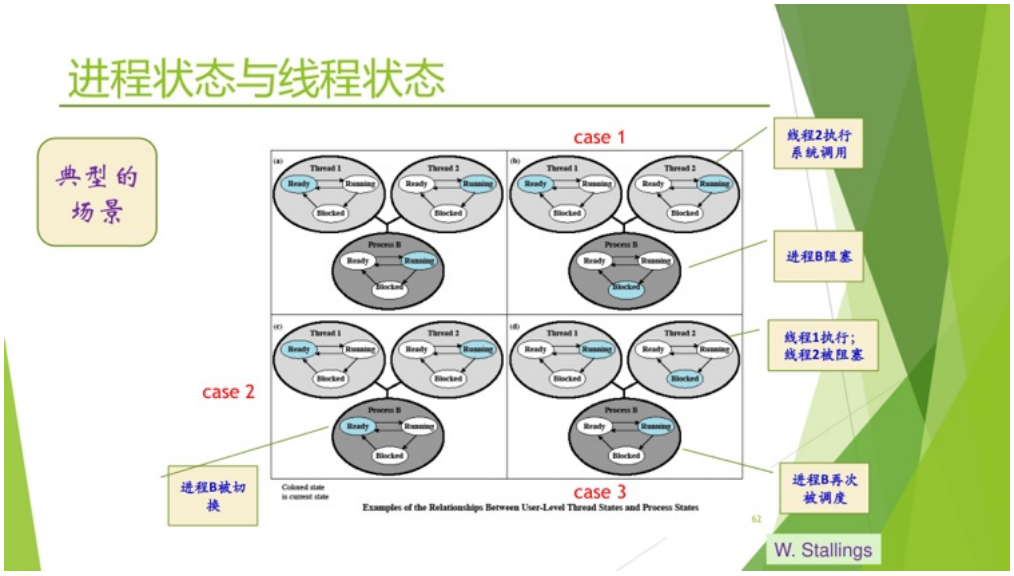
- case 2：B 中线程 2 执行阻塞系统调用 -> 整个进程 B 被内核阻塞；
- case 3：B 重新被调度回来。

POSIX Pthreads（IEEE 1003.4a）以线程库方式提供。早期 ULT 实现下，线程是**自愿**让出 CPU 的——内核不知道线程，无法剥夺。

ULT 优点：

- 切换快（不进入内核）；
- 调度算法可应用层定制；
- 可在不支持线程的 OS 上运行（只需要线程库）。

ULT 缺点：



图示：当线程 2 执行系统调用导致进程 B 阻塞时，进程 B 的所有线程都被一起阻塞。



图示：POSIX Pthreads 接口与“线程自愿让出 CPU”。

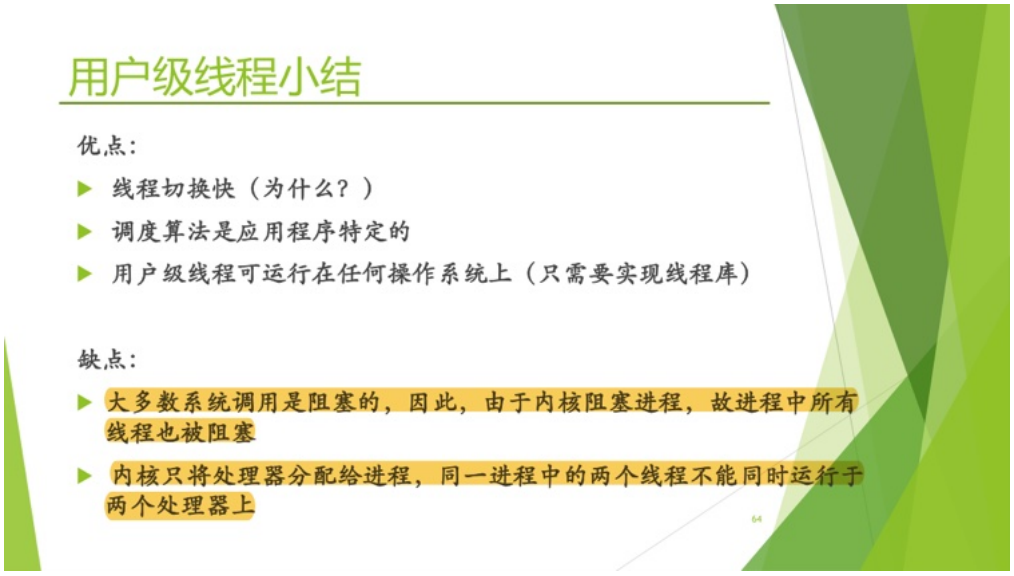
- 多数系统调用是阻塞的，一个线程阻塞 -> 整个进程阻塞；
- 内核只把 CPU 分给进程，同一进程的两个线程**不能同时**运行在两个核上。

阻塞系统调用问题

> **思考题：**如何缓解 ULT 的阻塞问题?

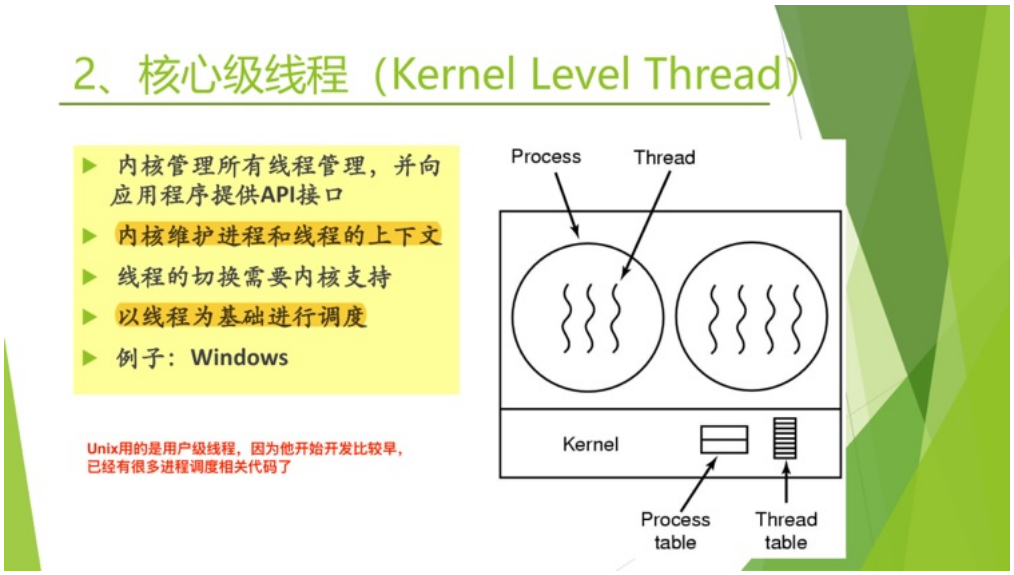
解答：两种方案：

1. **修改系统调用为非阻塞**：把可能阻塞的调用改成非阻塞 + 通知机制（如 `O_NONBLOCK + select / epoll`）；
2. **Jacketing / Wrapper 技术**：在线程库里给阻塞系统调用包一层壳：先用非阻塞方式探测；可能阻塞时，库切换到另一个就绪用户线程，等到事件完成再恢复。



图示：ULT 优缺点对比。

核心级线程 KLT



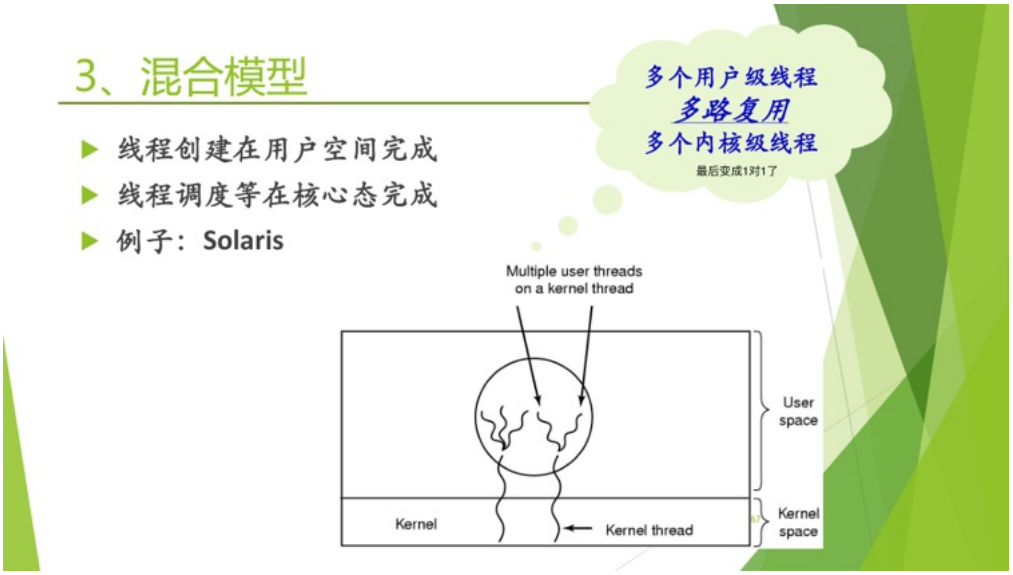
图示：内核管理所有线程，提供 API；调度以线程为单位。

- 内核管理所有线程，向应用程序提供 API；
- 内核维护进程上下文 + 每线程上下文；
- 线程切换需要内核支持；
- 调度以线程为基础进行；
- 例子：Windows、Linux NPTL（Native POSIX Thread Library）。

> 早期 UNIX 多采用 ULT，因为 UNIX 设计较早，已有大量进程调度代码。

混合模型

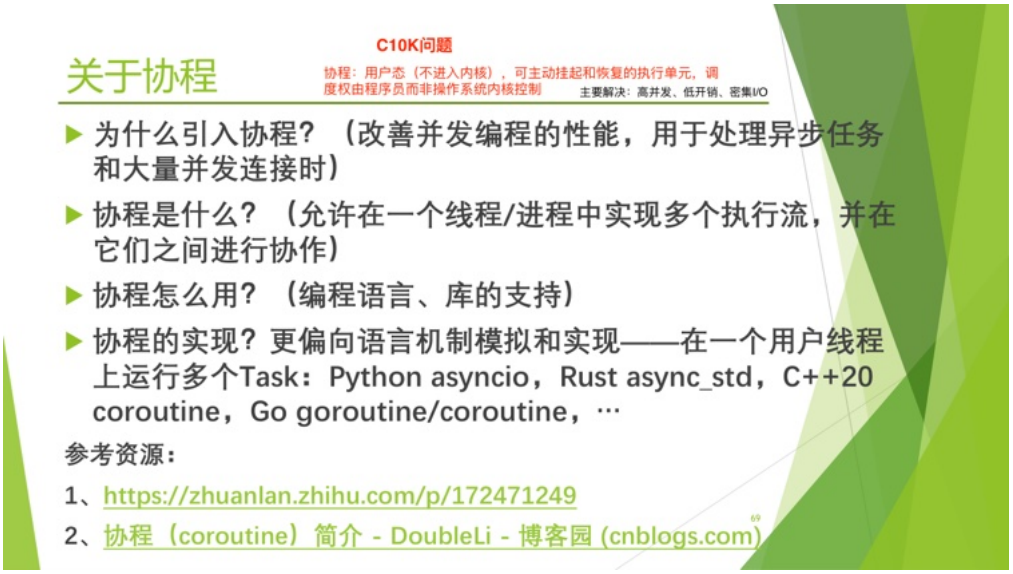
- 在用户空间创建线程，调度在内核态完成；



图示：M 个用户级线程映射到 N 个核心级线程，最终落到内核线程上。

- 多对一、一对一、多对多多种映射；
- 例子：早期 Solaris；后续主流系统多回退到 1:1（即每个用户线程对应一个内核线程）。

3.10 协程与纤程



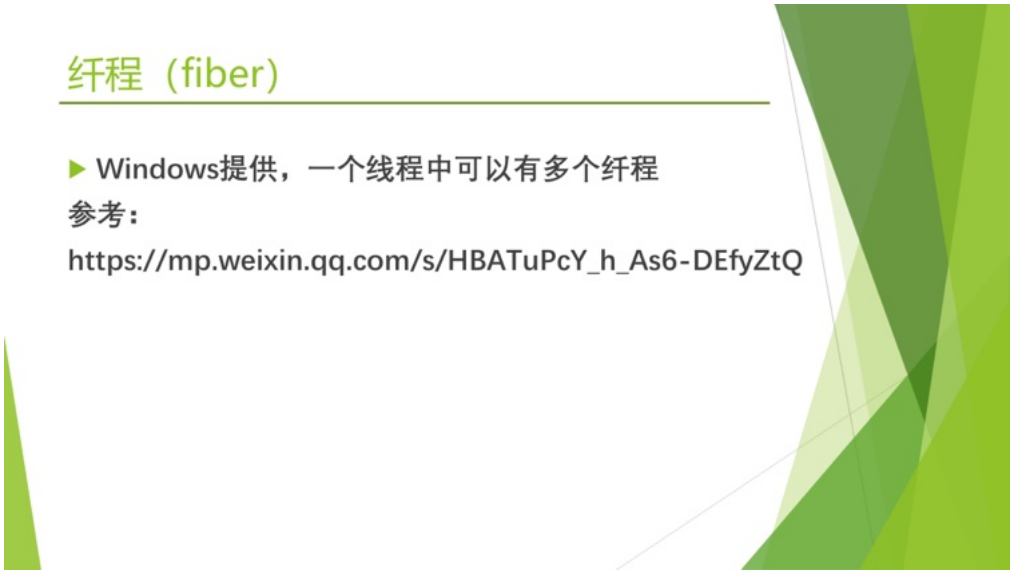
图示：协程是用户态的、可主动挂起 / 恢复的执行单元，调度由程序员控制。

- 协程出现是为了解决 **C10K** 类高并发问题：在不增加线程数的情况下，让单线程也能处理大量 I/O 任务。
- 协程不进入内核，切换开销极低；
- 主要适合 I/O 密集型场景：网络服务、爬虫、流量代理；
- 不适合 CPU 密集型场景（仍需多线程或多进程利用多核）。

> **要点：**“协程是用户态、可主动挂起和恢复的执行单元，调度权由程序员而非操作系统内核控制”。

使用方式：

- 编程语言原生：Python `async/await`、Go `goroutine`、C++20 `co_await/co_yield/co_return`、Rust `async/await`；
- 库支持：在不支持原生协程的语言里通过库实现；
- 模式：定义协程函数 -> 在 I/O 等待点 `yield` 让出 -> 事件循环 / 调度器恢复。



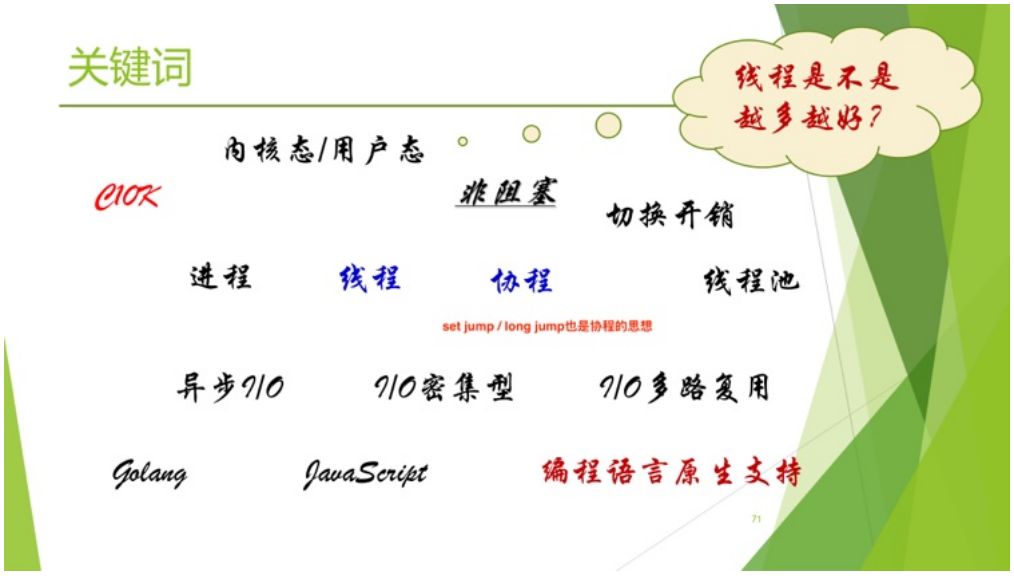
图示：Windows 的纤程（fiber）允许一个线程内有多个用户态调度单元。

- 纤程是 Windows 提供的“线程内可调度的协作执行单元”。
- 线程仍然由内核调度，但纤程间切换由用户代码控制。
- 与协程一样属于“用户态调度”，但 API 更接近 OS 而不是语言内置语法。

3.11 关键词与思维网络

记住几条主线：

- **执行实体的层次**：进程 -> 线程 -> 协程 / 纤程，逐级变轻、调度权逐级上移到用户。
- **资源 vs 调度**：进程负责资源、线程负责调度，分离后产生协程进一步细化。
- **常见话题**：
  - 异步 I/O 与 I/O 多路复用（`select / poll / epoll`）；
  - C10K 问题与高并发服务器；
  - 线程池（不是越多越好，要看 CPU / IO 比例）；
  - 切换开销 = 模式切换 + 缓存失效 + TLB 失效；
  - `setjmp / longjmp` 是 C 实现协程的底层机制。



图示：本讲关键词：进程、线程、协程、内核态/用户态、异步 I/O、I/O 密集型、Golang、JavaScript、线程池、C10K、I/O 多路复用、非阻塞、切换开销、setjmp/longjmp。

3.12 重点小结

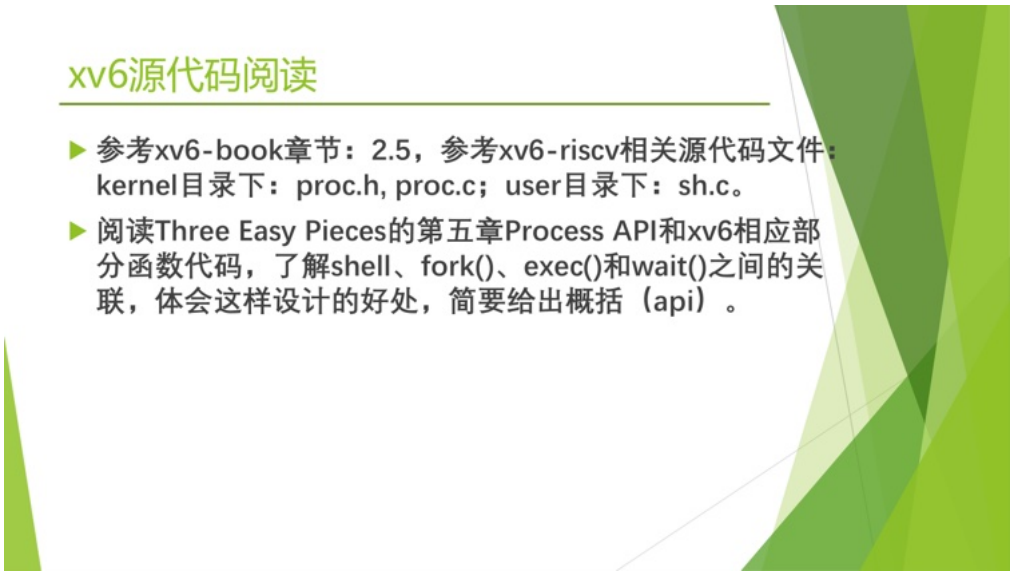


图示：本讲核心概念汇总。

- **进程的特性**：
  - **并发性**：进程可以同其他进程一起向前推进；
  - **动态性**：进程是“正在执行的程序实例”，动态产生、动态消亡；
  - **独立性**：进程是资源分配的独立单位；不同进程互不影响（地址空间相互独立）；
  - **制约性**：进程之间因共享资源 / 同步而产生制约；
  - **异步性**：每个进程以相对独立、不可预知的速度推进。
- **进程映像** = 程序 + 数据 + 栈 + PCB。
- **进程是可再入的（可重入）**：
  - **可再入程序**：可被多个进程同时调用的程序；

  - 必须是纯代码（不修改自身），调用者提供数据区。
- **线程**：
  - 多线程应用场景；
  - 线程基本概念、属性；
  - 三种线程实现机制；
- **协程**：
  - 用户态、协作式、密集 I/O 场景下的最佳工具。

3.13 XV6 源代码阅读



图示：本讲建议结合 xv6-book 第 2.5 节、`kernel/proc.{h,c}` 与 `user/sh.c` 阅读。

> **代码阅读题：**阅读 OSTEP 第 5 章 *Process API* 与 XV6 `kernel/proc.{h,c}`、`user/sh.c`，了解 shell、`fork()`、`exec()`、`wait()` 之间的关联，并简要给出概括。

解答：

1. `fork` (`kernel/proc.c`)：
   - 调用 `allocproc()` 找一个空闲 `struct proc` 槽，状态置为 `USED`；
   - 通过 `uvmcopy` 复制父进程的页表 / 内存（在 XV6 简单实现里直接复制，不是 COW）；
   - 复制 `trapframe`，把 `a0` 设为 0（子进程返回值）；
   - 共享父进程打开的文件（`filedup` 引用计数）、当前目录（`idup`）；
   - 拷贝父进程的 `name`；
   - 把子进程状态置为 `RUNNABLE`；
   - 父进程返回子进程 PID。
2. `exec` (`kernel/exec.c`)：

- 通过 namei 找到要执行的可执行文件；
- 解析 ELF 头、检查合法性；
- 为新地址空间分配新页表，按程序段加载到内存；
- 在新地址空间中分配两页栈：一页保护页（不可访问），一页用户栈；
- 把命令行参数和环境拷贝到用户栈；
- 替换 proc->pagetable、proc->sz、proc->trapframe->epc 等；
- 从 exec 返回时，trapframe 已指向新程序入口。

3. **wait (kernel/proc.c)**：
   - 父进程查找处于 ZOMBIE 状态的子进程；
   - 找到则收集其退出状态、释放其页表与内核栈、把状态置回 UNUSED；
   - 没找到则 sleep 在 wait_lock 上，等待 exit 唤醒。
4. **exit (kernel/proc.c)**：
   - 关闭所有打开文件、释放当前目录引用；
   - 把子进程的父指针指向 init（避免成孤儿）；
   - 设置 xstate，把状态置为 ZOMBIE，调用 wakeup(parent)；
   - 调用 sched() 把控制权交给调度器，进程不再返回。
5. **shell (user/sh.c)**：
   - 主循环：getcmd -> parsecmd -> runcmd；
   - runcmd 根据命令类型派生子进程：
     - 简单命令：fork() + exec(argv[0], argv)；
     - 重定向：fork() 后子进程 close(fd) + open(...)，再 exec；
     - 管道：pipe() + fork() + fork() + dup2，左右两半串联；
     - 后台命令 &：父进程不 wait；
   - 父进程通过 wait(0) 等待子进程结束。

设计概括：

- fork + exec + wait + exit 四件套是 UNIX 进程模型的核心；
- 它们职责单一、可正交组合，使 shell 的实现保持紧凑；
- 这是“**机制简单、组合强大**”的 UNIX 哲学的代表案例。

LECTURE 4

第 4 讲

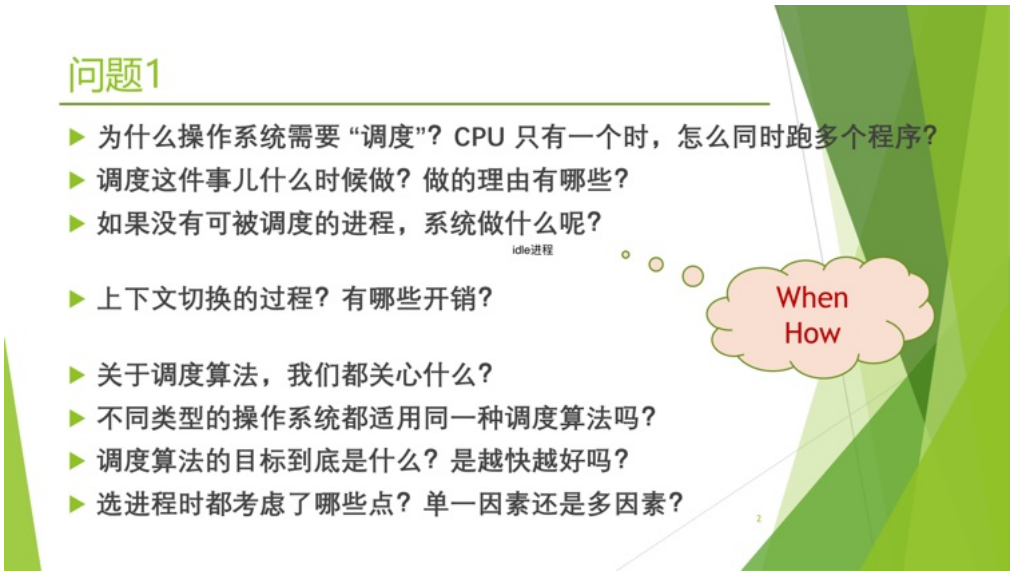
进程线程调度

Operating Systems Course Notes

4.1 本讲导读

- 这一讲解决一个核心问题：**多个就绪进程 / 线程争用 CPU 时，操作系统该怎么决策？**
- 围绕三件事展开：
  - **What**：用什么算法选下一个执行实体；
  - **When**：什么时候做选择；
  - **How**：选好之后怎样把 CPU 真正交给它（上下文切换）。
- 知识脉络：
  1. 调度三层次、调度时机与上下文切换；
  2. 调度算法的设计目标和指标；
  3. 经典算法（FCFS / SJF / SRTN / HRRN / RR / 优先级 / 多级反馈队列）；
  4. Windows、Linux、Solaris、openEuler 的真实调度器；
  5. 多核 / 实时调度等扩展话题。

4.2 三组开篇思考题



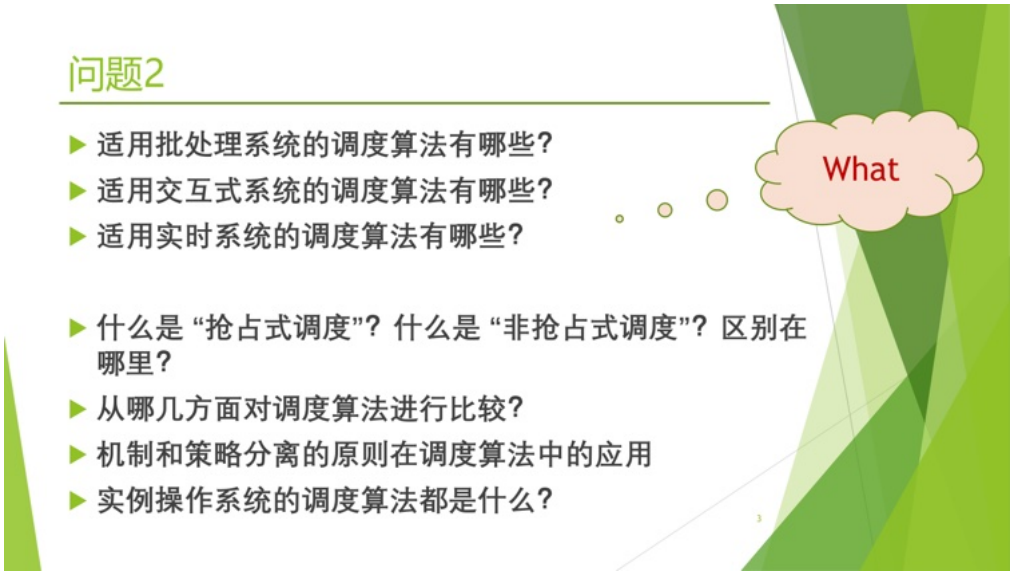
图示：调度的必要性、时机、上下文切换、衡量指标等关键问题。

下面给出全部思考题的精炼解答。

> **思考题 1：**为什么操作系统需要调度？只有一个 CPU 时怎么同时跑多个程序？

解答：

- 物理 CPU 数量有限，但系统中常有大量就绪进程 / 线程。
- 操作系统通过把 CPU 时间切成片段，并让它们在 CPU 上轮流推进，使每个执行实体都觉得自己拥有独占的 CPU。
- 这种“**对 CPU 时间的合理分配**”就是调度。



图示：批处理 / 交互式 / 实时各自适用的算法、抢占式与非抢占式、机制 / 策略分离。

> **思考题 2：**调度什么时候做？有哪些理由？

解答：调度通常发生在以下事件后、内核返回用户态之前：

- 进程创建 / 终止；
- 运行进程主动 `yield` 或调用阻塞性系统调用；
- 时钟中断（时间片到、计时器到时）；
- I/O 中断（被等待的事件发生，使阻塞进程变就绪）；
- 异常 abort、新进程被唤醒。

这些事件都会让“就绪队列”发生变化，因此必须重新决定下一个上 CPU 的进程。

> **思考题 3：**没有可调度的进程时系统做什么？

解答：调度器会让 CPU 运行**空闲进程（idle process）**：

- 优先级最低，只在没有任何用户/系统进程就绪时执行；
- 通常在循环中执行 `HLT` 等指令把 CPU 置入低功耗状态；
- 可顺便完成轻量级的系统维护工作。

> **思考题 4：**上下文切换的过程？有哪些开销？

解答：见 §3 详细展开；要点是“**直接开销 + 间接开销**”：

- 直接：保存 / 恢复寄存器、切换页表（指令昂贵、cycle 多）；
- 间接：缓存（L1/L2/L3）、Buffer Cache、TLB 失效；新进程刚开始执行时 cache miss 率很高。

> **思考题 5：**关于调度算法，我们关心什么？

解答：

- **性能指标**：吞吐量（Throughput）、周转时间（Turnaround Time）、响应时间（Response Time）；
- **非性能指标**：公平性（Fairness）、资源利用率（CPU Utilization）、强制优先级、平衡资源；
- **特定场景指标**：实时性（截止期限、可预测性）、能耗。
- 这些指标常常彼此冲突，需要权衡。

> **思考题 6：**不同类型 OS 都用同一种调度算法吗？

解答：不会。

系统类型	主要目标	常用算法
批处理	吞吐量、CPU 利用率、平均周转时间	FCFS、SJF、SRTN、HRRN
交互	响应时间、公平性	RR、优先级、多级反馈队列、CFS
实时	截止期限、可预测性	速率单调（RM）、最早截止时间优先（EDF）

> **思考题 7：**调度算法的目标到底是什么？是越快越好吗？

解答：调度不是单一指标优化。

- 批处理系统强调吞吐量；
- 交互系统强调响应时间和公平性；
- 实时系统强调可预测的截止期限；
- 通用系统在各项之间折中。
- “**越快越好**”往往牺牲公平、可预测性或交互响应，因此并不普遍适用。

> **思考题 8：**选进程时考虑了哪些因素？单一还是多因素？

解答：

- 单一因素：FCFS（到达时间）、SJF（预测的下次 CPU burst 长度）、简单优先级。
- 多因素：现代 OS（Linux CFS、Windows 抢占式优先级、MLFQ）通常混合多种特征：优先级、等待时间、CPU 行为、I/O 倾向、前后台等。

> **思考题 9：**什么是抢占式？什么是非抢占式？区别是什么？

解答：见 §6.5 详细展开。要点：

- **非抢占式**：CPU 一旦交给进程，除非它自己释放，否则不被剥夺；实现简单、开销小，但响应差。

- **抢占式**：内核可以根据时钟中断或更高优先级进程的就绪强制切换；响应好、能保证公平和实时性，但需要处理临界区与同步。

> **思考题 10：**从哪几方面对比调度算法？

解答： CPU 利用率、吞吐量、周转时间、等待时间、响应时间、公平性、可预测性、算法本身的开销与实现复杂度、对参数（时间片、优先级）的敏感性。

> **思考题 11：**机制 / 策略分离怎样体现在调度里？

解答：

- **机制**：上下文切换、PCB、就绪队列结构、时钟中断处理、`yield` / `wakeup` 原语。
- **策略**：调度算法本身（FCFS / RR / 优先级 / CFS）、时间片大小、优先级动态调整方式。
- Linux 通过 `sched_setscheduler()` 让用户在多种策略中切换，但底层的上下文切换机制是共用的。

> **思考题 12：**不同实例 OS 的调度算法是什么？

解答：

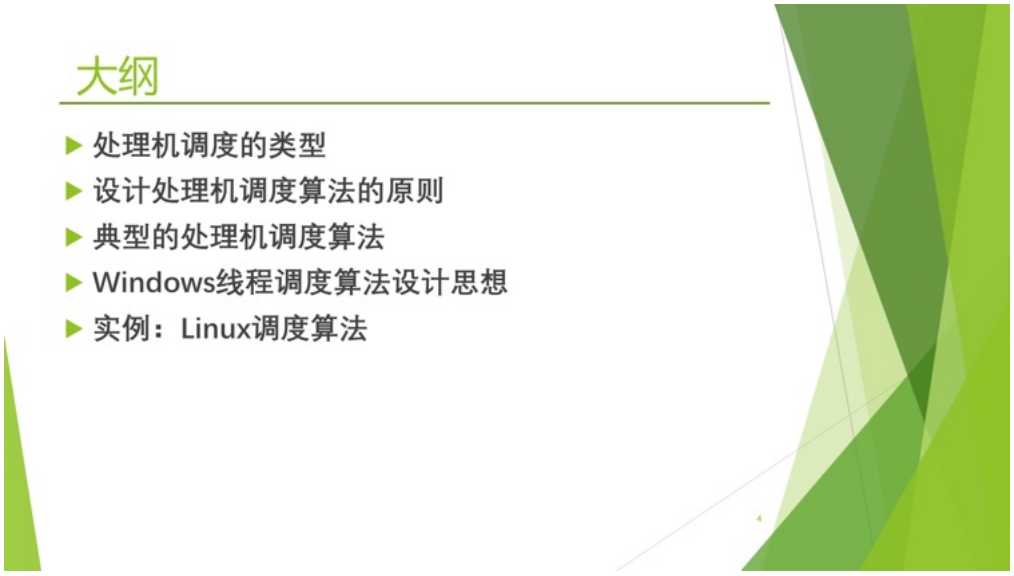
- UNIX：动态优先数法；
- 4.3 BSD：多级反馈队列；
- Windows：基于优先级的抢占式多任务调度 + 时间配额调整；
- Linux：抢占式调度，普通进程用 CFS / EEVDF，实时进程用 `SCHED_FIFO` 或 `SCHED_RR`；
- Solaris：综合调度算法。

后面 §10、§11 详细展开。

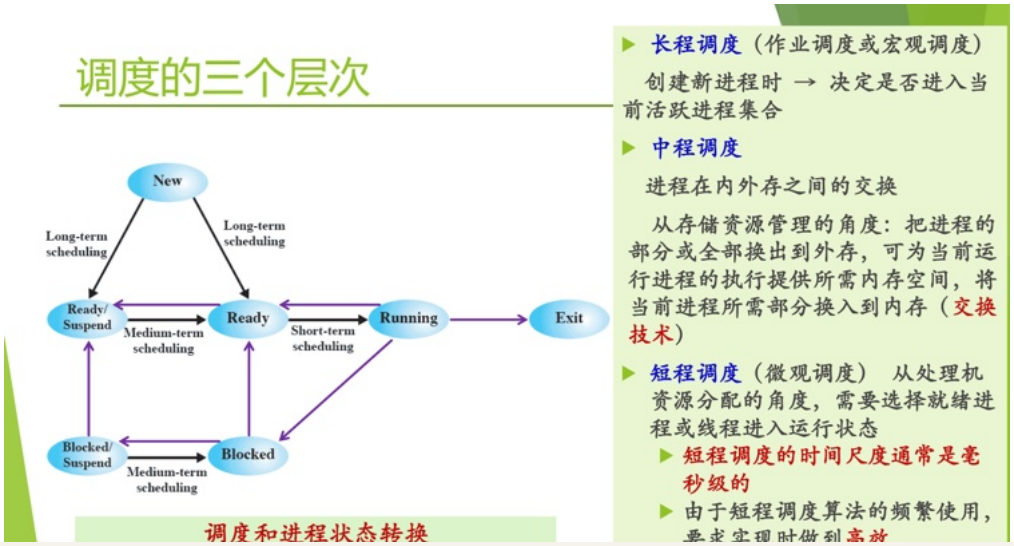
4.3 大纲与三个层次

4.3.1 调度的三个层次

层次	时机	决策内容	时间尺度
长程调度（作业调度 / 宏观调度）	创建新进程	是否进入“当前活跃进程集合”，控制并发度	较慢
中程调度（内存调度）	内存紧张	把进程的全部或部分换出到外存（挂起）或换回	中等
短程调度（CPU 调度 / 微观调度）	中断 / 异常 / 系统调用返回前	从就绪队列选下一个执行实体	毫秒级



图示：调度类型 -> 设计原则 -> 典型算法 -> Windows 线程调度 -> Linux 调度。



> **要点：**短程调度发生最频繁，因此要求高效。本讲讨论的“调度算法”默认指短程调度。

4.4 短程调度的三个问题：What / When / How

短程调度要解决三个问题：

- **What**：依据什么原则挑选进程 / 线程？-> 调度算法（§5–§9）。
- **When**：什么时候做选择？-> 调度时机（§3.1）。
- **How**：怎样把 CPU 真正交给它？-> 上下文切换（§3.2）。

4.4.1 CPU 调度的时机

典型时机：

- 进程执行完毕并退出；



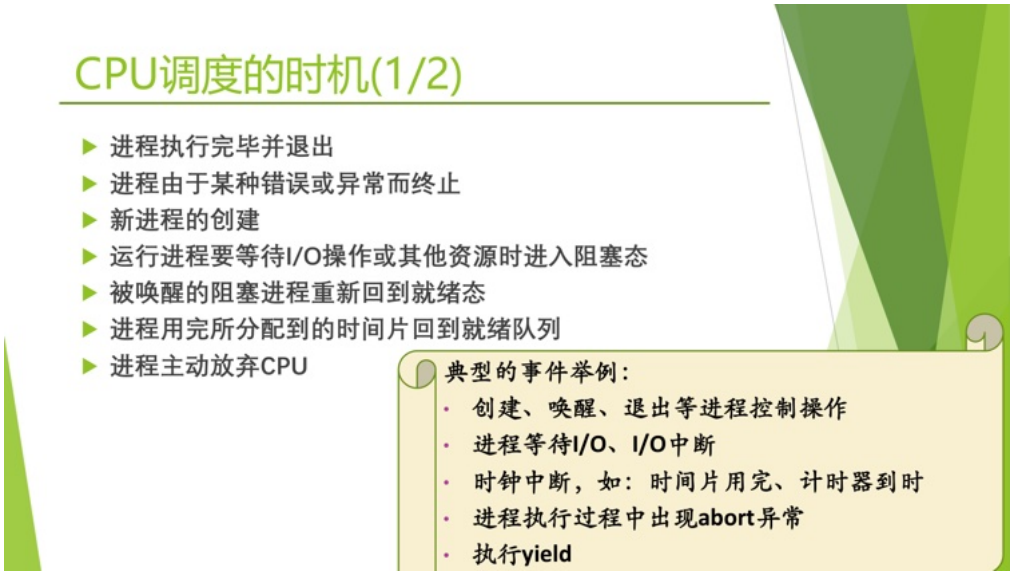
图示：N 个就绪进程，M 个 CPU，按调度算法决定谁上谁的哪个 CPU。



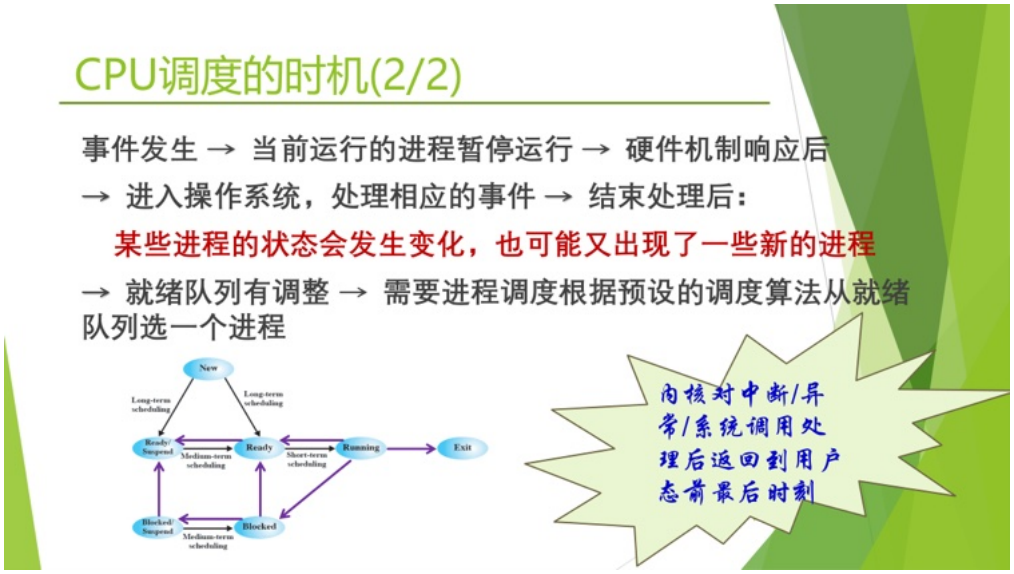
图示：What（依据何原则）、When（何时分配）、How（怎样切换）。

- 进程因错误 / 异常被终止；
- 创建新进程；
- 运行进程要等待 I/O 或其他资源进入阻塞态；
- 阻塞进程被唤醒回到就绪态；
- 进程用完时间片回到就绪队列；
- 进程主动 `yield` 让出 CPU。

> **底层规律：**任何让“就绪队列发生变化”的事件都可能引发调度；事件本身通过中断 / 异常 / 系统调用进入内核，调度发生在事件处理结束、返回用户态之前。



图示：进程退出 / 异常终止 / 创建 / 阻塞 / 唤醒 / 时间片用完 / 主动让出。



图示：调度通常在内核处理完事件、即将返回用户态前的最后时刻。

4.4.2 调度过程：进程切换

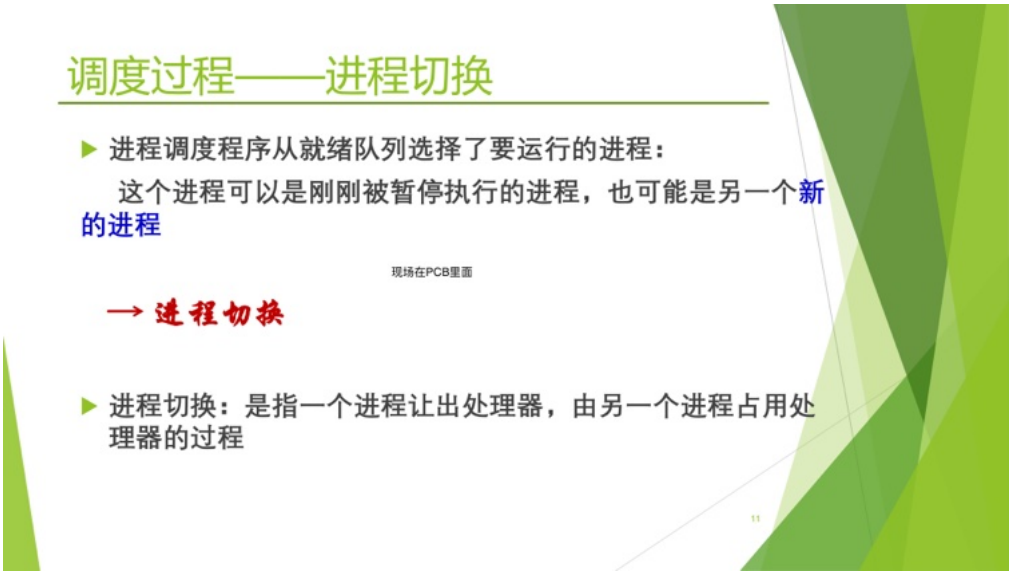
- 调度程序选好下一个进程后，就要执行**进程切换 / 上下文切换**。
- 这个进程可以是刚被暂停的进程（例如时间片到期重新选回它），也可以是另一个新的进程。

上下文切换

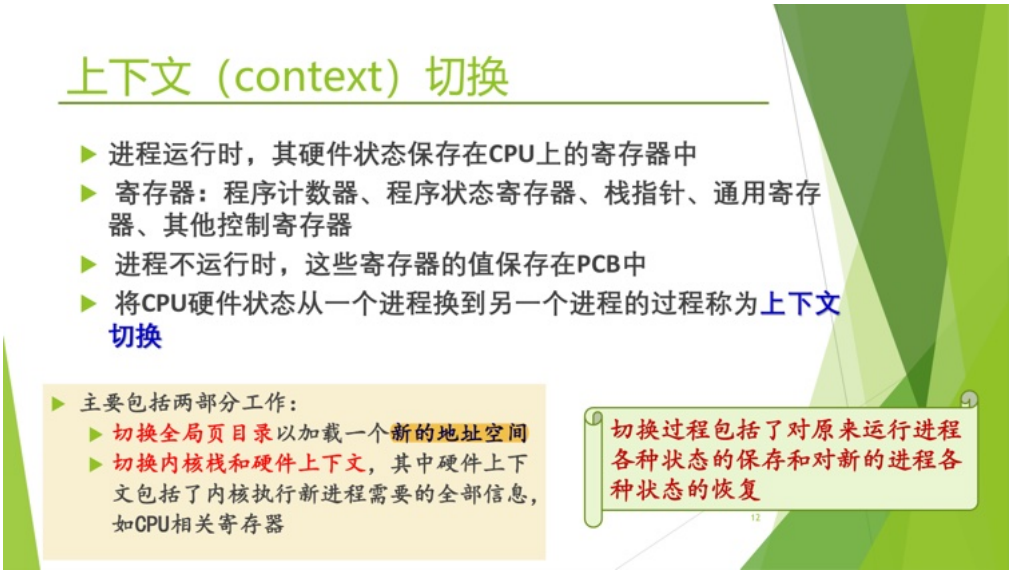
上下文切换主要包括两部分工作：

1. 切换全局页目录，加载新地址空间；
2. 切换内核栈和硬件上下文，包括 CPU 相关寄存器。

具体步骤（A 下 CPU、B 上 CPU）：



图示：进程让出 CPU、由另一进程占用 CPU 的过程。现场保存在 PCB 中。

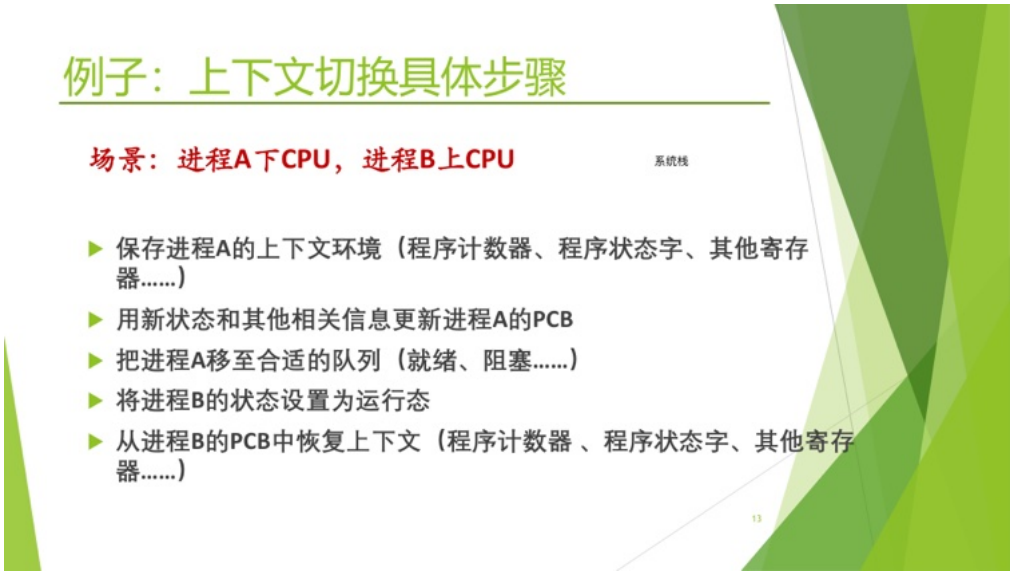


图示：上下文 = 寄存器 + PSW + 栈指针；切换时要保存原进程上下文，恢复新进程上下文。

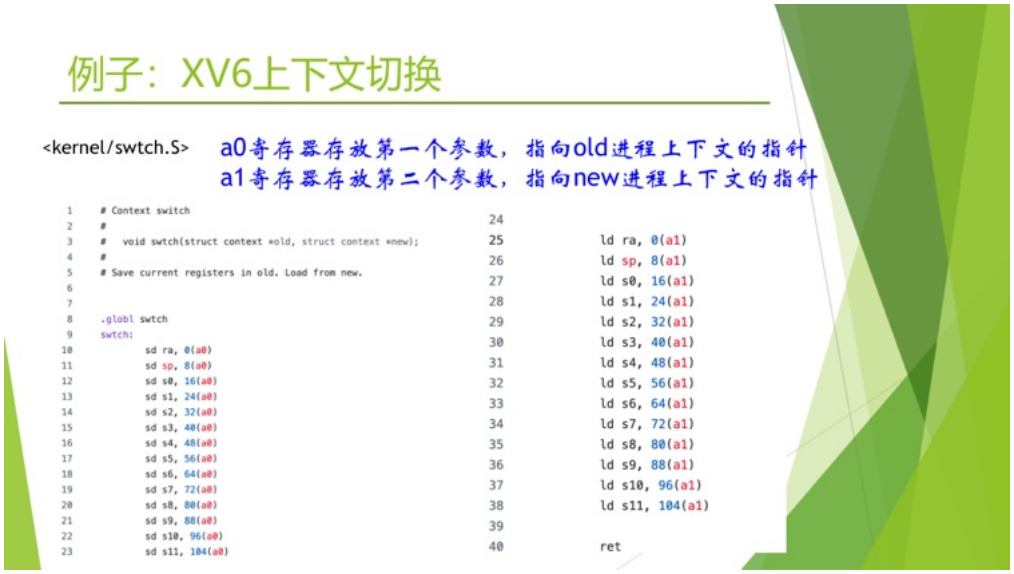
1. 保存进程 A 的上下文（PC、PSW、寄存器...）；
2. 用新状态 / 信息更新 A 的 PCB；
3. 把 A 移入合适队列（就绪、阻塞）；
4. 把进程 B 的状态置为运行；
5. 从 B 的 PCB 恢复上下文。

XV6 上下文切换

```
.globl swtch
swtch:
    sd ra,   0(a0)
    sd sp,   8(a0)
```



图示：保存 A 上下文 -> 更新 A 的 PCB -> 把 A 移入合适队列 -> B 状态置运行 -> 从 B 的 PCB 恢复上下文。



图示：`swtch(old, new)`：第一个参数 a0 指向 old context，第二个 a1 指向 new context。

```
    sd s0,  16(a0)
    /* ...保存 s1~s11... */
    ld ra,   0(a1)
    ld sp,   8(a1)
    ld s0,  16(a1)
    /* ...恢复 s1~s11... */
    ret
```

要点：

- 只保存被调用者保存寄存器（callee-saved），其余寄存器由编译器约定保留；
- `ra` 是返回地址；切换后 `ret` 直接跳到新进程上次保存的执行位置；
- 所有 RISC-V 寄存器都通过 `proc->context` 维护，配合 `kstack` 完成切换。

上下文切换开销



图示：直接开销与间接开销。

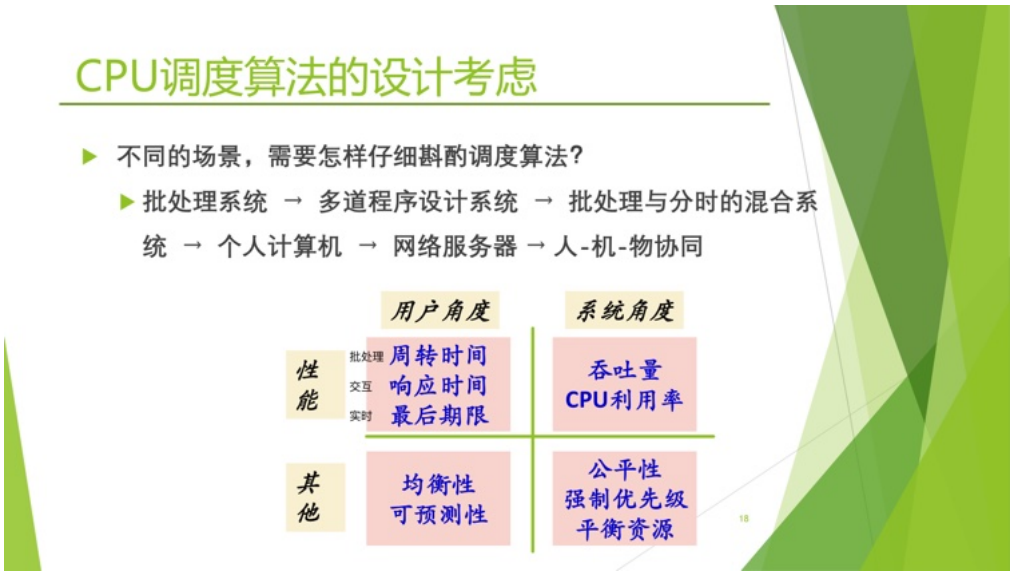
- **直接开销**：内核完成切换所用的 CPU 时间（保存/恢复寄存器、切换地址空间，相关指令较昂贵）。
- **间接开销**：高速缓存（Cache）、缓冲区缓存（Buffer Cache）、TLB 失效；新进程开始执行时 cache miss 多。
- **结论**：上下文切换不能太频繁，时间片太小会被切换开销吃光。

4.5 调度算法的设计目标



图示：交互式进程、批处理进程、实时进程对应的不同目标。

进程类型	关键目标	例子
交互式	响应时间快、平均延迟低于 50–150 ms、均衡性	shell、文本编辑、GUI
批处理	吞吐量、周转时间、CPU 利用率	营业额计算、利息计算、索赔处理
实时	截止期限、可预测性	视频/音频、机械控制



图示：从用户角度（性能 + 其他）到系统角度（吞吐量 / 利用率），需要在批处理 / 交互 / 实时之间斟酌。

设计调度算法时要权衡的维度：

- 用户角度（性能）：周转时间、响应时间、最后期限、可预测性；
- 用户角度（其他）：公平性、强制优先级、均衡性；
- 系统角度：吞吐量、CPU 利用率、平衡资源。

4.5.1 调度算法的衡量指标

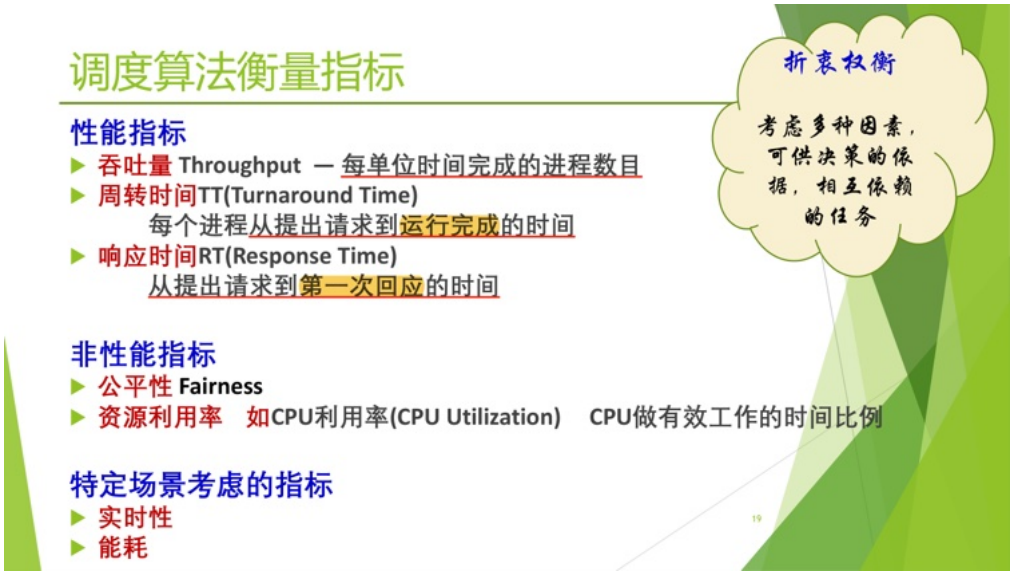
- **吞吐量（Throughput）**：单位时间完成的进程数。
- **周转时间（Turnaround Time, TT）**：进程从提出请求到运行完成的时间。
- **响应时间（Response Time, RT）**：从提出请求到第一次回应的时间。
- **CPU 利用率**：CPU 做有效工作的时间占比。
- **公平性**：所有进程都能获得合理的 CPU 时间份额。
- **特定场景**：实时性、能耗。

这些指标常常相互冲突，因此选择算法是“折衷权衡（trade-off）”的过程。

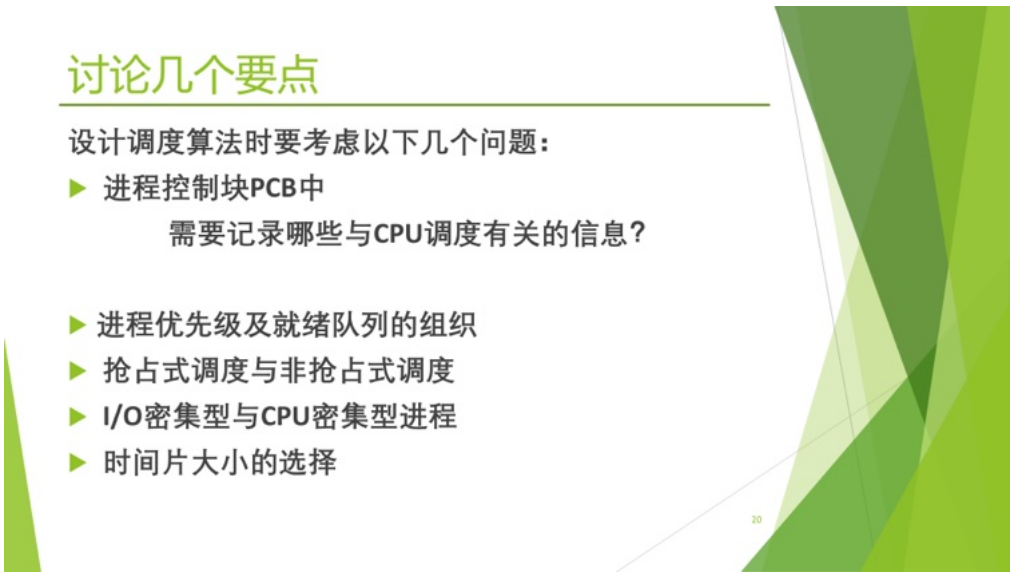
4.6 调度算法设计的若干要点

4.6.1 进程优先级（数）

- **静态优先级**：进程创建时指定，运行过程中不变；



图示：吞吐量、周转时间、响应时间、CPU 利用率、公平性。



图示：PCB 中需记录的调度信息、优先级、就绪队列组织、抢占性、I/O / CPU 密集型进程、时间片选择。

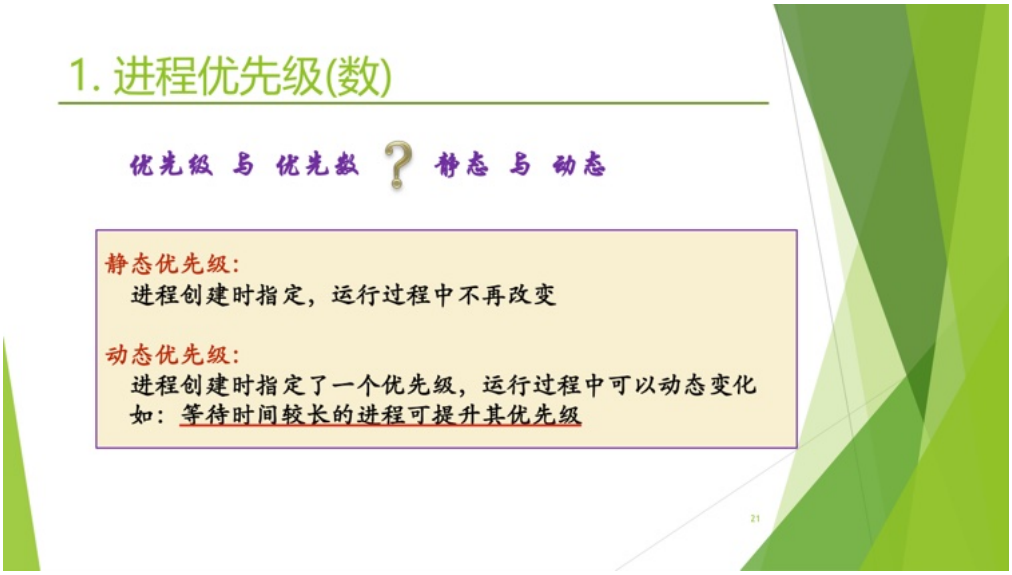
- **动态优先级**：可在运行中调整，例如：
  - 等待时间过长 -> 提升优先级（避免饿死）；
  - I/O 等待结束 -> 临时提升优先级（提升交互响应）。

4.6.2 就绪队列组织

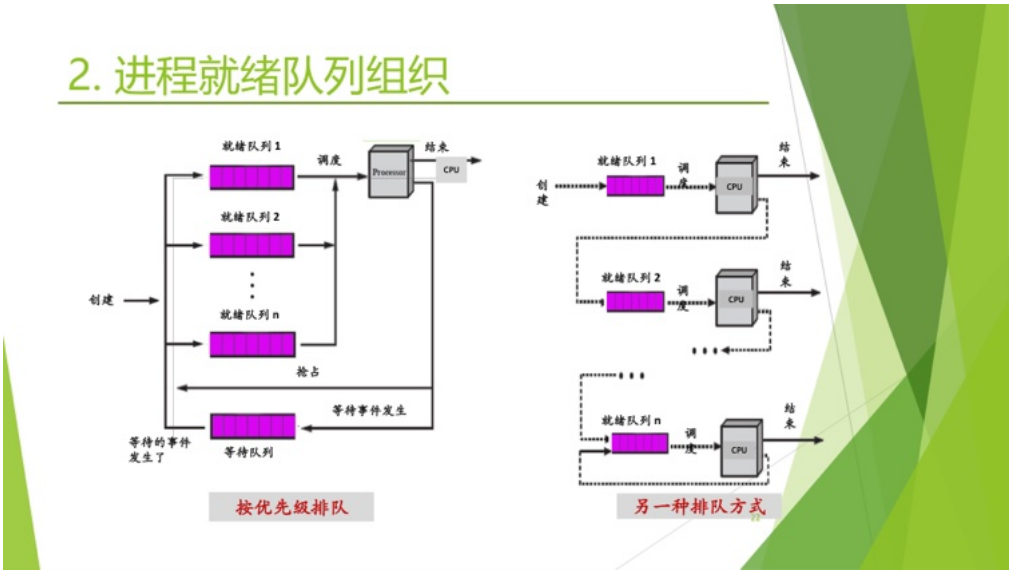
两种典型方式：

1. **单队列按优先级排序**：插入时按优先级有序插入；
2. **多队列（每优先级一个队列）**：调度时先按优先级选队列，再在队列内按 FIFO 或 RR 选进程。

第二种是 Linux O(1) 与 Windows 调度器的基础。



图示：静态优先级与动态优先级。



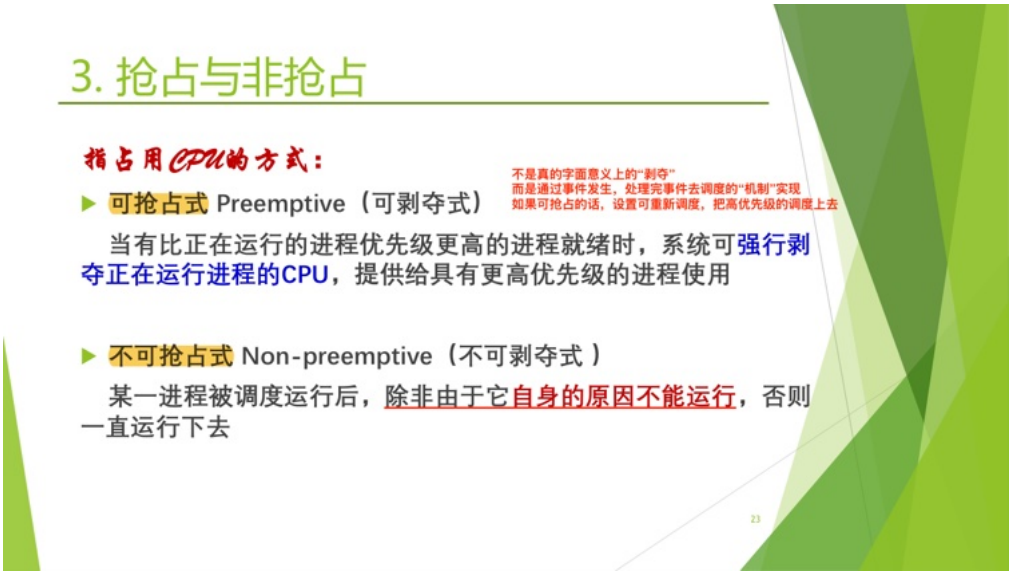
图示：单队列按优先级排队 vs. 每个优先级一个队列。

4.6.3 抢占式 vs 非抢占式

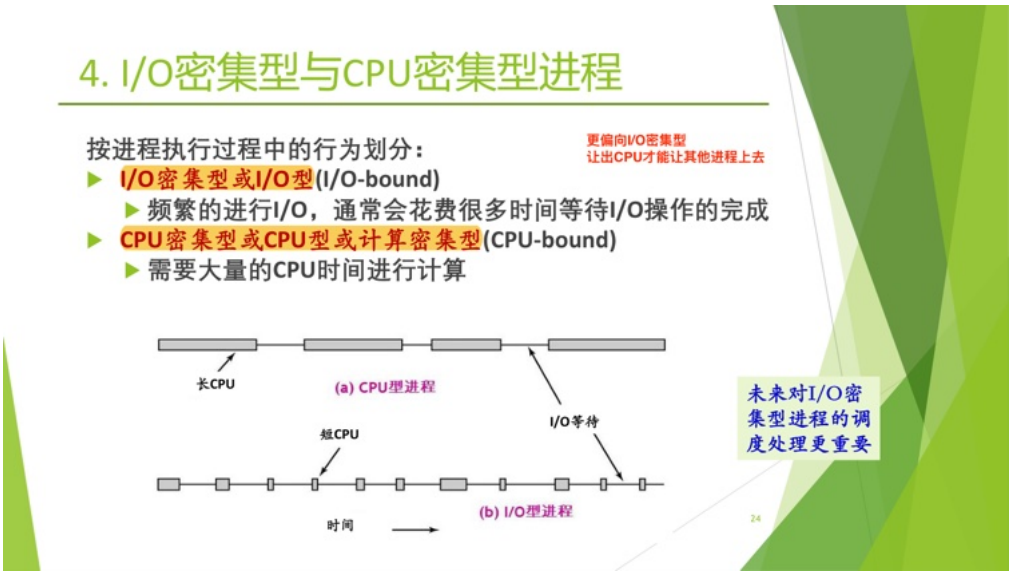
- **可抢占式 Preemptive**：当出现更高优先级的就绪进程，系统可强行剥夺当前 CPU。
- **不可抢占式 Non-preemptive**：除非进程自身阻塞或退出，否则一直运行。
- 现实里“剥夺”不是字面意义上抢，而是通过事件发生（时钟中断、唤醒）触发重新调度，把高优先级的进程切上去。

4.6.4 I/O 密集型 vs CPU 密集型

- **I/O 密集型（I/O-bound）**：频繁 I/O，CPU 使用率低，但响应敏感。
- **CPU 密集型（CPU-bound）**：长时间计算，对 I/O 不敏感。
- 现代调度倾向于偏好 I/O 密集型进程：



图示：抢占强调的是“可剥夺”，但实际通过事件驱动的重新调度实现。



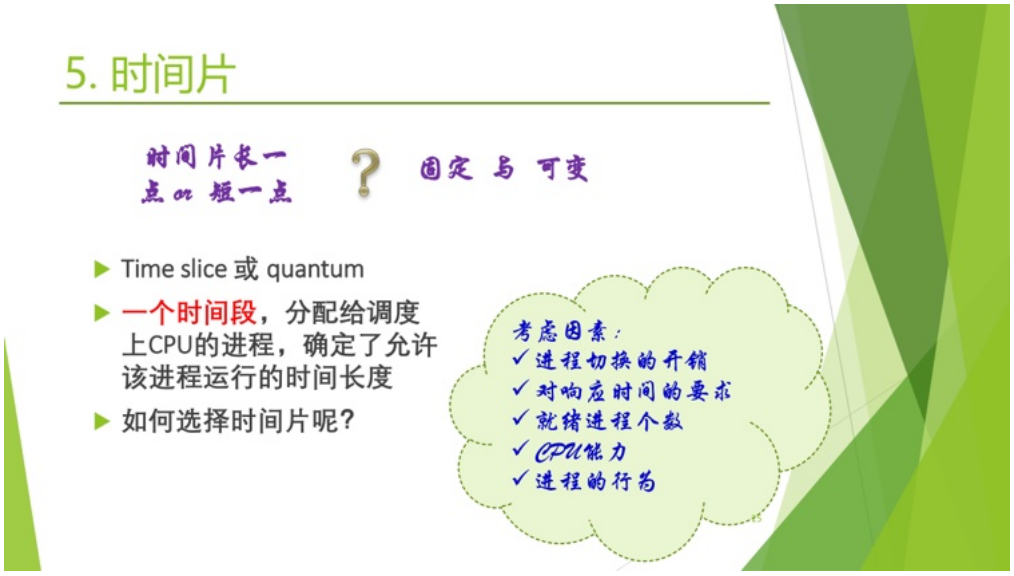
图示：I/O 型进程多次短 CPU + I/O 等待；CPU 型进程长 CPU 计算。

- 让出 CPU 才能让其他进程上去；
- I/O 完成后及时唤醒，提高交互体验。

4.6.5 时间片

考虑因素：

- 进程切换开销；
- 响应时间要求；
- 就绪进程个数；
- CPU 能力；
- 进程的行为（I/O 还是 CPU 密集）。



图示：时间片长度的选择需要权衡进程切换开销、响应时间、CPU 能力等因素。

时间片可以是固定也可以是可变（如 Windows 给前后台不同时间片）。

4.7 批处理系统中的调度算法



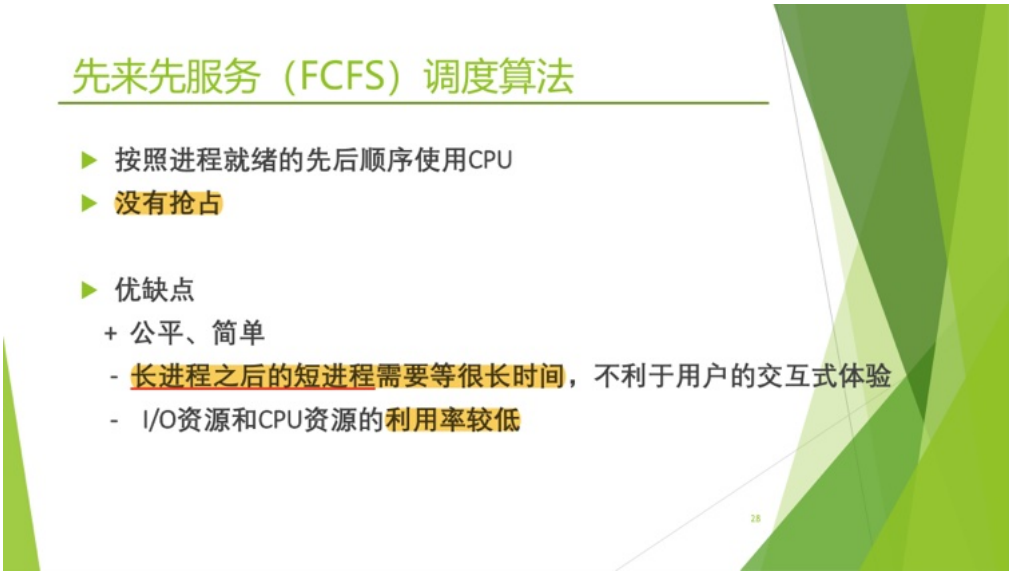
图示：FCFS / SJF / SRTN / HRRN，目标是吞吐量、周转时间、CPU 利用率与公平。

4.7.1 先来先服务 FCFS

- **算法**：按进程就绪先后顺序使用 CPU，无抢占。
- **优点**：公平、简单。
- **缺点**：长进程之后的短进程要等很久（**护航效应 Convoy Effect**）；I/O 与 CPU 利用率低。

例子

调换执行顺序：



图示：FCFS 按到达顺序服务，无抢占；公平简单但护航效应严重。

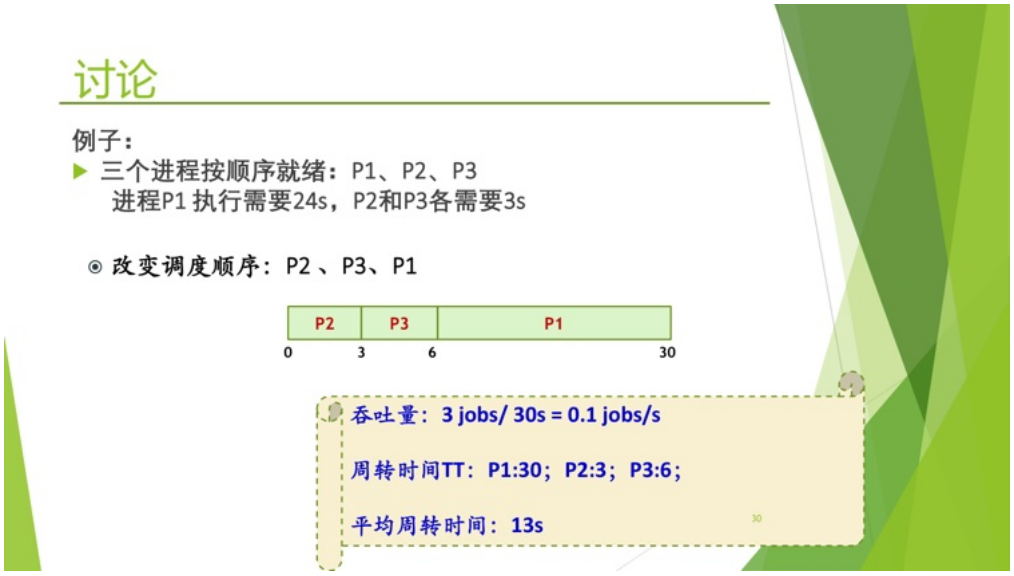


图示：P1=24s、P2=3s、P3=3s 顺序服务，平均周转时间 27s。

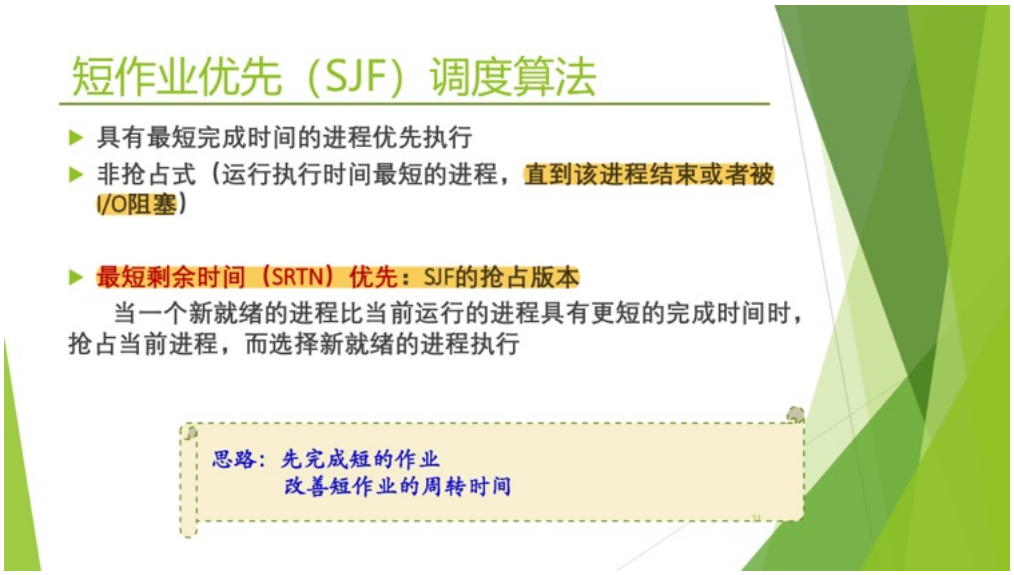
> **要点：** 周转时间 `T(turnaround) = T(completion) - T(arrival)`。

4.7.2 短作业优先 SJF / SRTN

- **SJF（Shortest Job First）**：选下一个 CPU burst 最短的进程，非抢占。
- **SRTN（Shortest Remaining Time Next）**：抢占式 SJF，新进程比当前剩余时间短就抢占。



图示：先 P2、P3 再 P1，平均周转时间降到 13s，说明 FCFS 对顺序敏感。



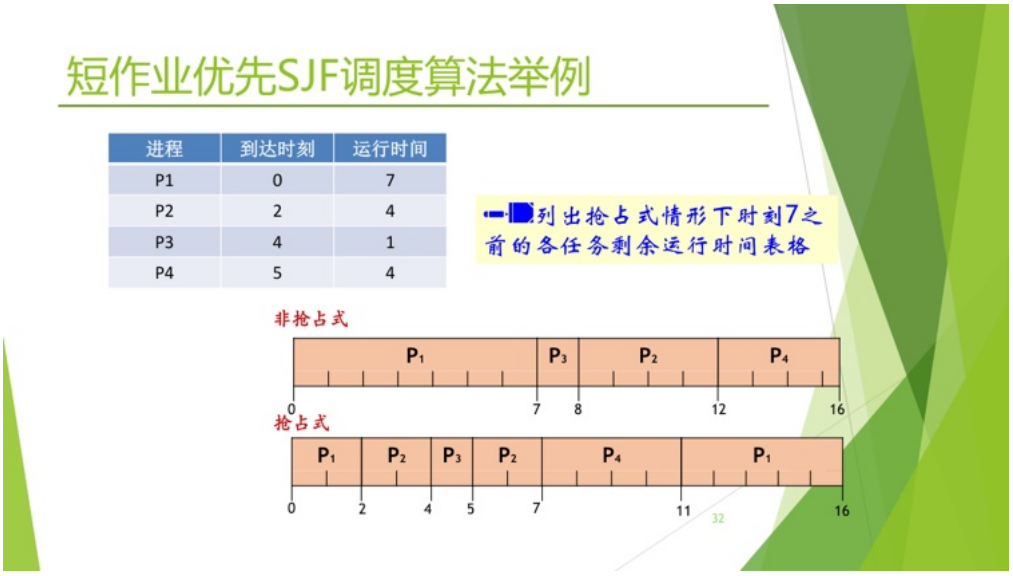
图示：SJF 选最短完成时间的进程，非抢占；SRTN 是 SJF 的抢占版。

例子

进程	到达时刻	运行时间
P1	0	7
P2	2	4
P3	4	1
P4	5	4

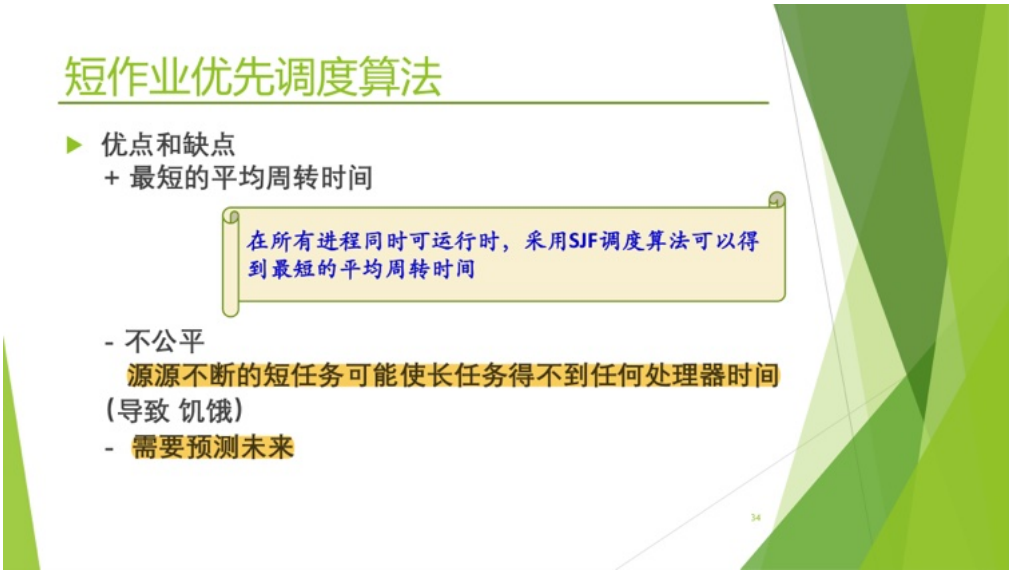
- 非抢占 SJF：P1 -> P3 -> P2 -> P4
- 抢占 SRTN：P1 -> P2 -> P3 -> P2 -> P4 -> P1

> **小练习：**抢占式情况下时刻 7 之前各任务剩余运行时间表（自己列一遍以巩固）。



图示：四个进程在不同时刻到达，非抢占与抢占两种 SJF 的执行序列对比。

优缺点



图示：最短平均周转时间 / 不公平 / 可能饿死 / 需要预测未来。

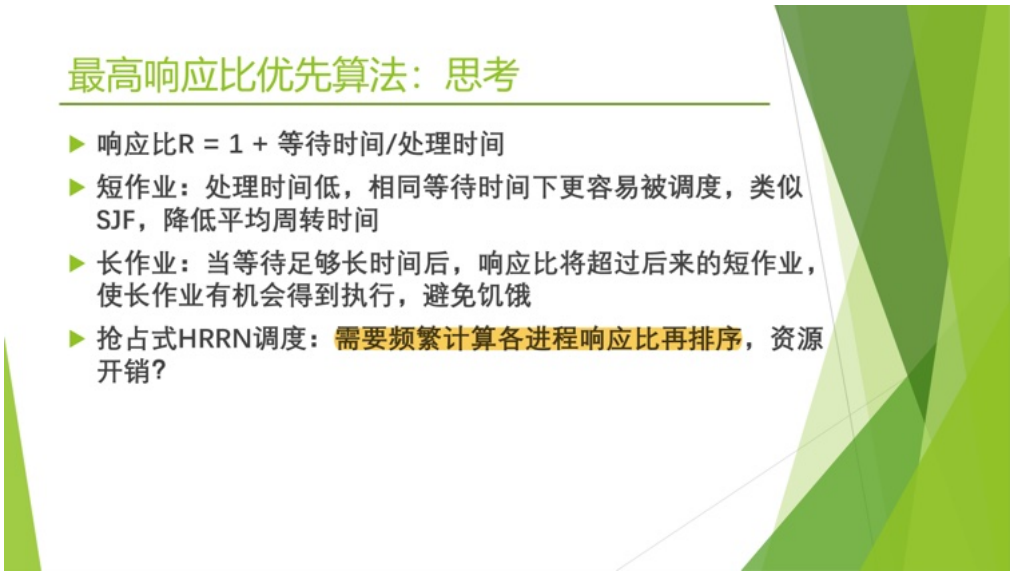
- 优点：在所有进程同时可运行时，SJF 给出**最短平均周转时间**。
- 缺点：
  - 不公平；
  - 长任务可能被源源不断的短任务饿死；
  - 需要预测下一个 CPU burst 长度（实际中只能基于历史平均估计）。

4.7.3 最高响应比优先 HRRN

- **响应比 R = 周转时间 / 处理时间 = 1 + 等待时间 / 处理时间**。
- 总是选 R 最高的进程。



图示：响应比 R = (作业等待时间 + 处理时间) / 处理时间 = 1 + 等待时间 / 处理时间。



图示：短作业天然 R 大；长作业等待久后 R 也会上升，避免饿死。

- 短作业处理时间小、相同等待下 R 大，类似 SJF；
- 长作业等到一定时间后 R 上升，避免饿死。
- 抢占式 HRRN 需要频繁计算响应比，开销大；通常只用非抢占式。

4.8 交互式系统中的调度算法

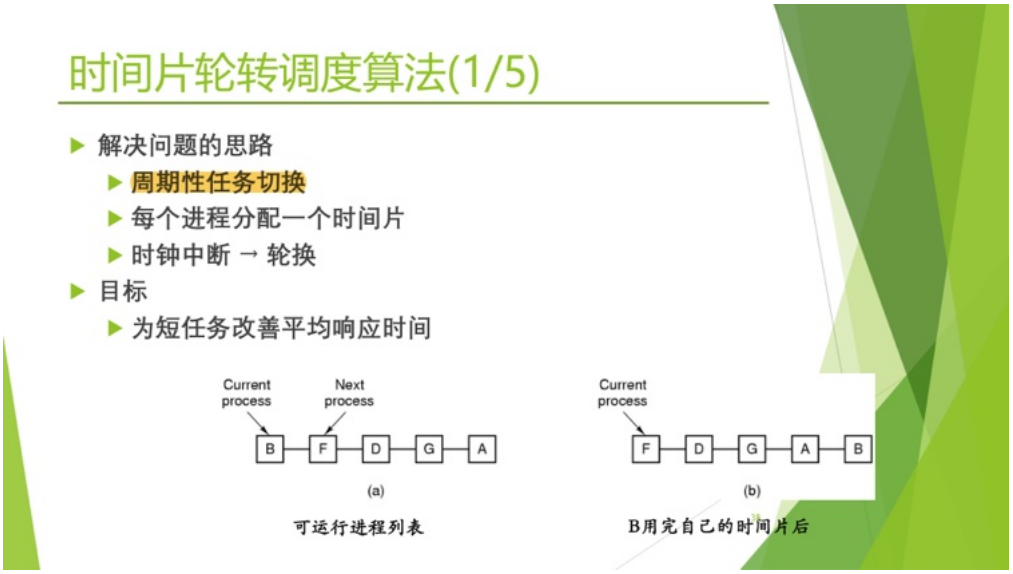
4.8.1 时间片轮转 RR

- 每个进程被分配一个时间片，到期由时钟中断触发切换。
- 目的：为短任务提供良好响应时间。

例子



图示：RR、HPF、多级队列、多级反馈队列、SPN 等。



图示：每个进程分配一个时间片，时钟中断时切换；目标是改善短任务平均响应时间。

时间片大小的影响

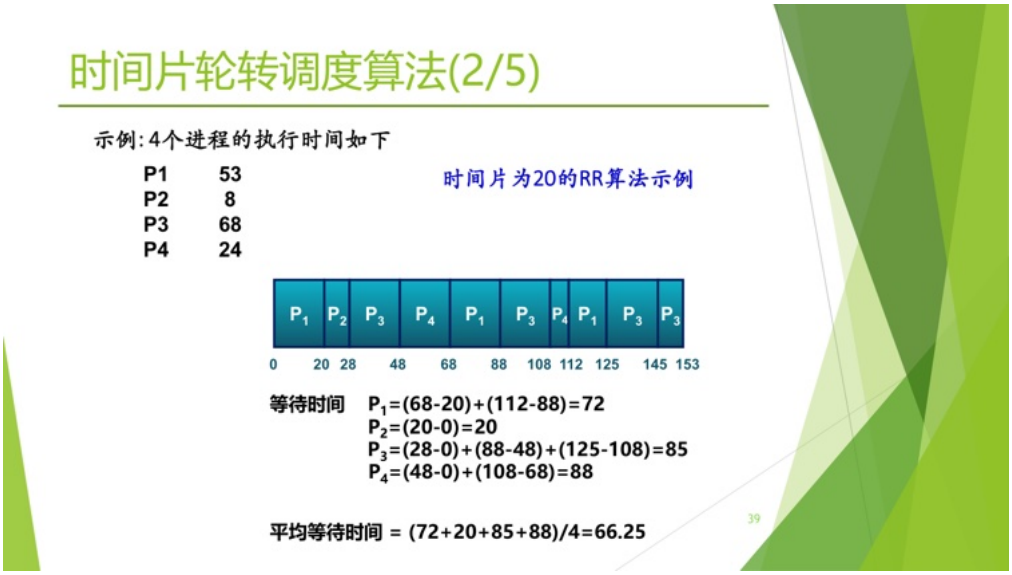
- 太长 -> 退化为 FCFS，响应变差；
- 太短 -> 切换开销吃掉 CPU；
- 现实中常选“略大于平均交互时间”。

优缺点

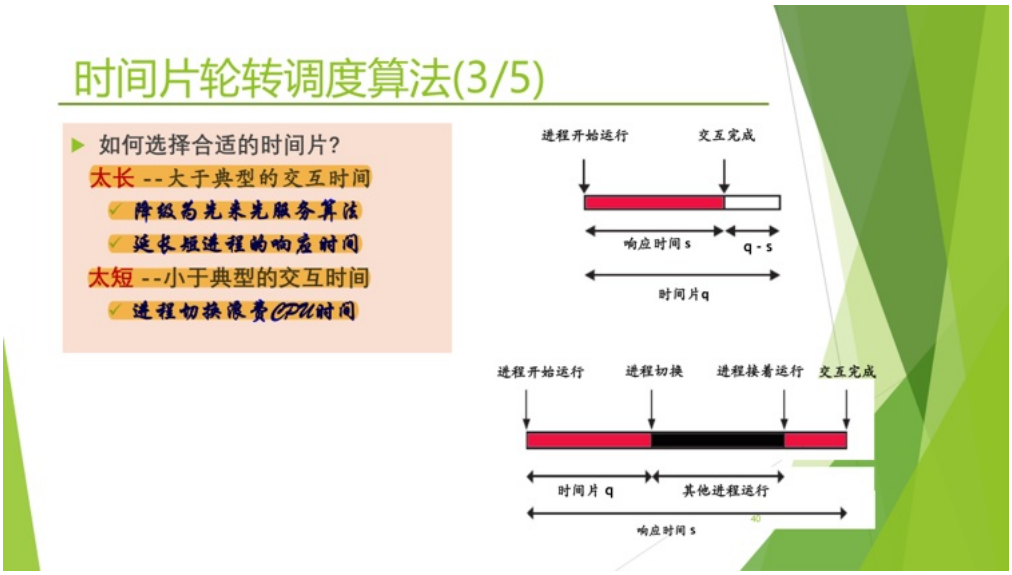
- RR 对短任务友好；
- 对长度相同的任务可能不利（在多次切换中拖长平均完成时间）。

FCFS 与 RR 的全面对比

要点：



图示：4 个进程在时间片 20 下的 RR 调度过程，平均等待时间 66.25。



图示：时间片太长退化为 FCFS，太短切换开销大。

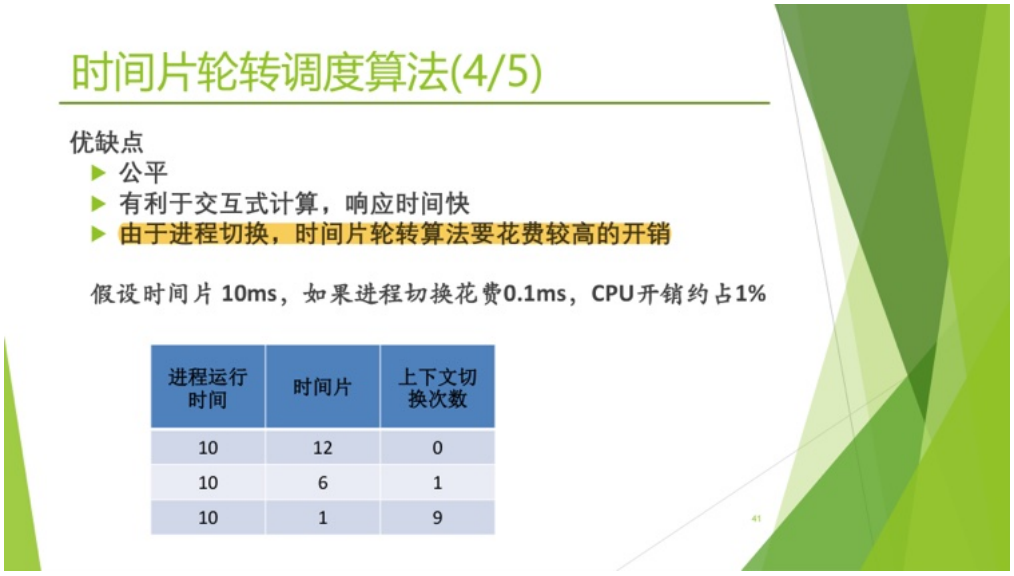
- BestFCFS（短先长后）平均等待时间 31.25；
- WorstFCFS（长先短后）平均等待时间 83.5；
- RR 在不同时间片下波动，但通常居中。

4.8.2 虚拟轮转法 Virtual RR

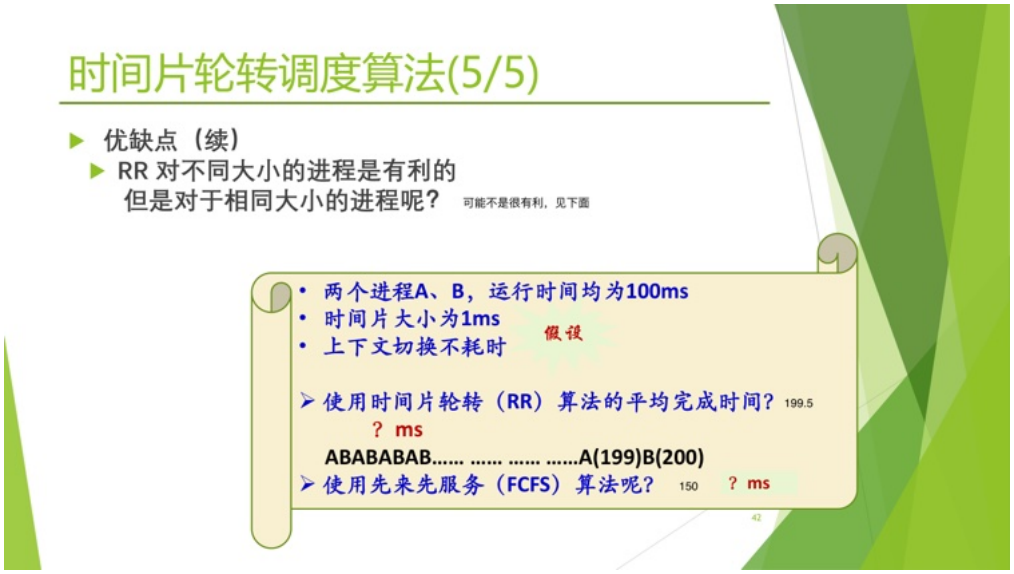
VRR 通过引入辅助队列：

- I/O 阻塞后重新就绪的进程进入辅助队列；
- 调度时优先从辅助队列取任务并补偿其剩余时间片；
- 解决了传统 RR 中 I/O 型进程响应慢的问题。

例子：



图示：公平、对交互友好、切换开销随时间片变小而升高。



图示：对相同长度任务不友好的反例：100ms × 2 个进程，时间片 1ms，RR 平均完成 199.5ms vs FCFS 的 150ms。

4.8.3 优先级调度

- 优先级可以静态或动态；
- 就绪队列按优先级组织；
- 实现简单；
- 缺点：低优先级可能饿死（starvation）；
- 缓解：老化（aging）、动态优先级提升。

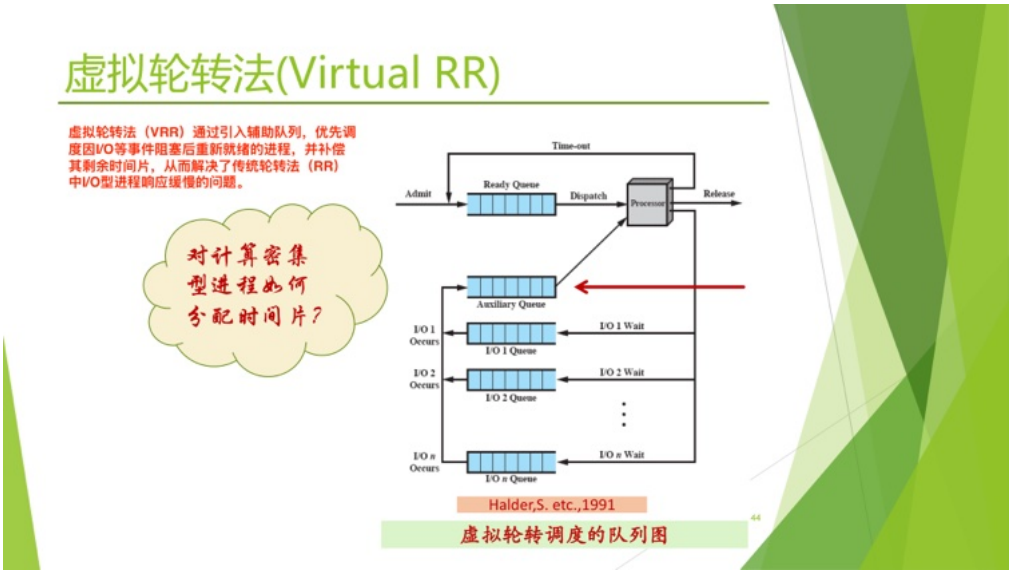
4.8.4 优先级反转

- **现象**：高优先级 H 因等待低优先级 L 持有的资源被阻塞；同时 L 又被中优先级 M 抢占，导致 H 间接被 M 阻塞。



时间片	P_1	P_2	P_3	P_4	平均等待时间
RR(q=1)	84	22	85	57	62
RR(q=5)	82	20	85	58	61.25
RR(q=8)	80	8	85	56	57.25
RR(q=10)	82	10	85	68	61.25
RR(q=20)	72	20	85	88	66.25
BestFCFS	32	0	85	8	31.25
WorstFCFS	68	145	0	121	83.5

图示：四个进程在不同时间片大小下 RR 与 FCFS 的平均等待时间对比。



图示：用辅助队列优先调度由 I/O 阻塞回到就绪态的进程，并补偿其剩余时间片。

- **影响**：可能引发系统错误、降低系统性能。
- **解决方案**：
  - **优先级天花板（Priority Ceiling Protocol）**：临界资源记录其最高使用者优先级；持有它的进程在临界区内提升到该优先级。
  - **优先级继承（Priority Inheritance）**：H 等待 L 的资源时，L 临时继承 H 的优先级。
  - **使用中断禁止**：在极短的临界区内禁中断（要谨慎）。

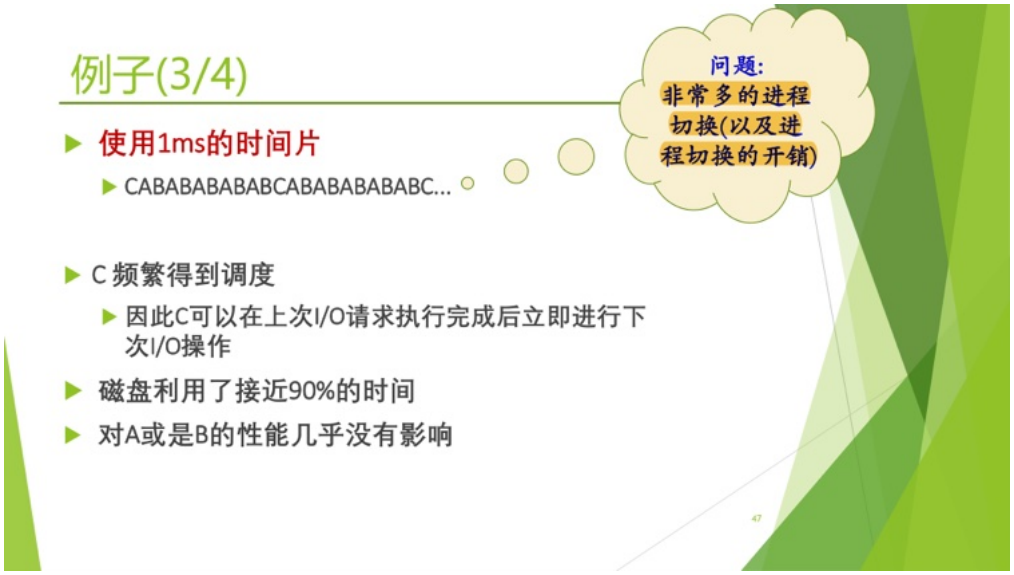
4.8.5 多级队列调度

设计要点：

- 按什么分队列（前后台、用户/系统、CPU/IO 型等）；
- 队列内调度算法（高优先级常用 RR，低优先级用 FCFS）；



图示：A、B 各需 1000ms 计算；C 反复执行 1ms 计算 + 10ms I/O。



图示：时间片为 1ms 的 RR 让 C 频繁调度，磁盘利用率近 90%。

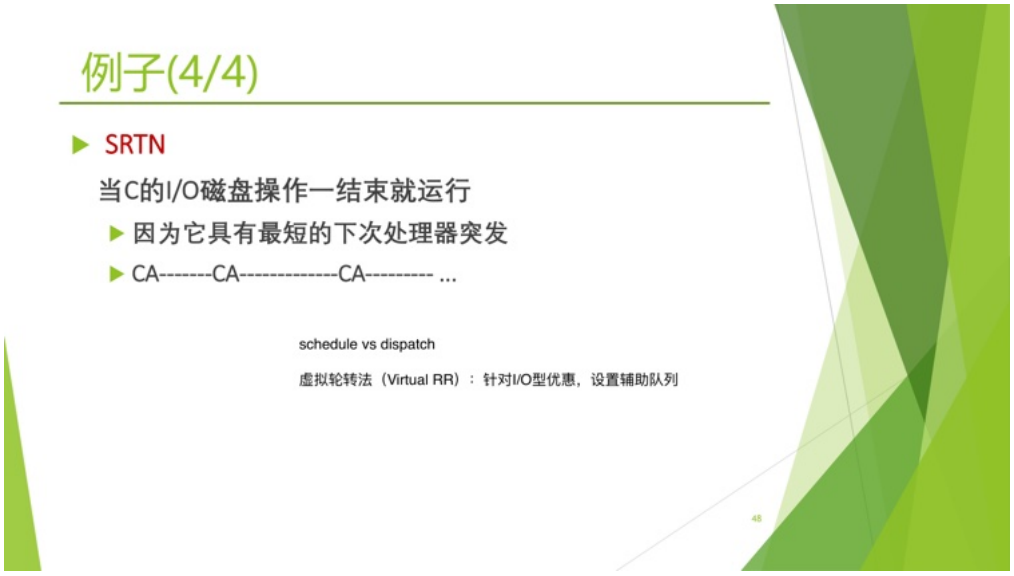
- 是否允许进程在队列间移动；
- 队列之间的优先级如何决定。

> **小细节：**一种偏好 I/O 型的做法是“时间片没用完就被打断（被 I/O 阻塞）的进程回来时不降级”，体现了对交互型的倾斜。

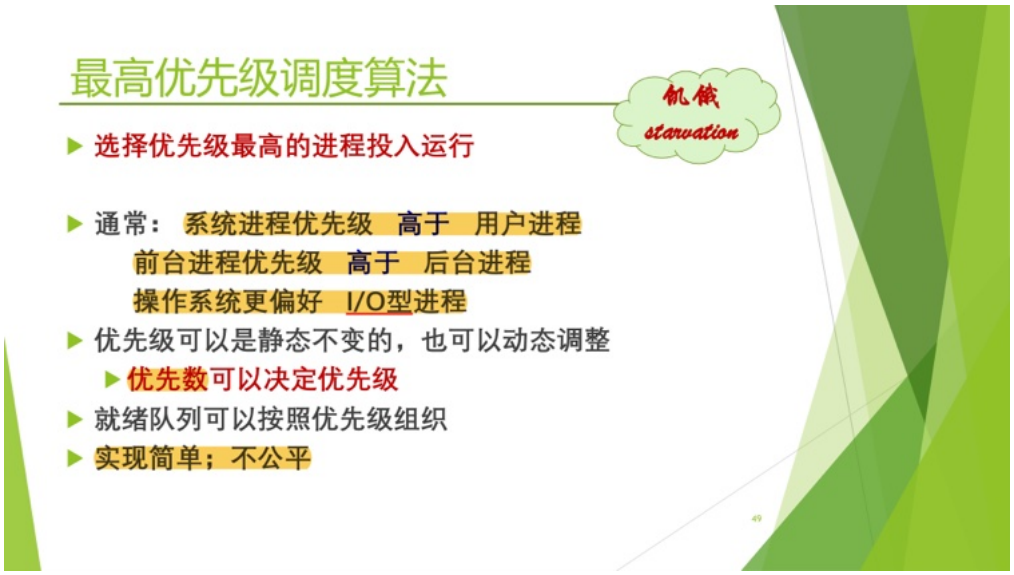
4.8.6 多级反馈队列 MLFQ

非抢占版本规则：

- 设置多级就绪队列，第 1 级优先级最高；
- 各级时间片不同，第 1 级最小，逐级增大；
- 当第 1 级为空时调度第 2 级，以此类推；



图示：SRTN 让 C 一旦 I/O 完成立刻执行；schedule vs dispatch 的差异显现。



图示：选优先级最高的进程；系统进程 > 用户进程，前台 > 后台，OS 偏好 I/O 型。

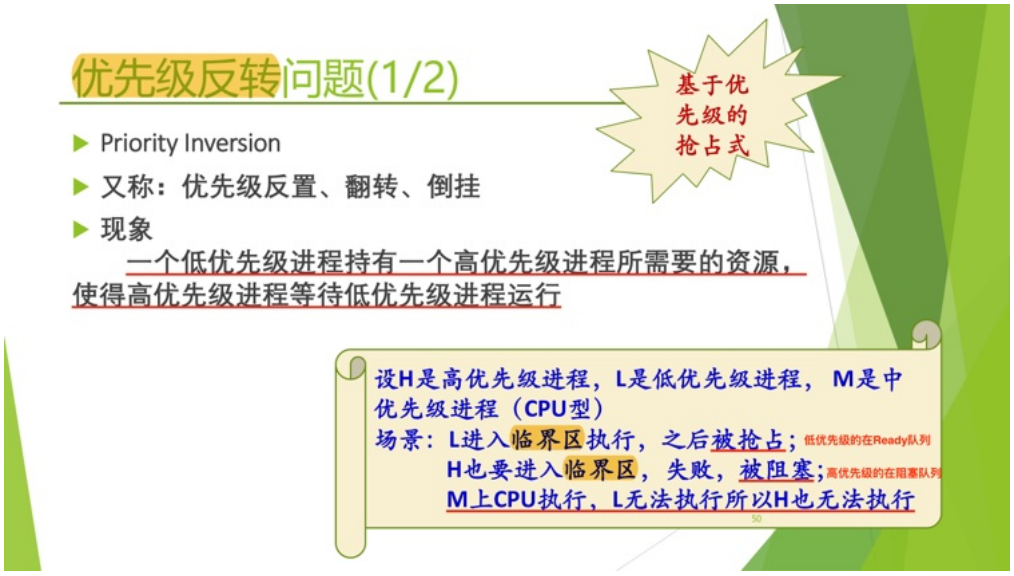
- 各级队列内按 RR 调度；
- 新进程进入第 1 级；
- 用完时间片放弃 CPU 的进程进入下一级；
- 因阻塞放弃 CPU 的进程在事件发生后回到原级（队首 / 队尾的选择见 §7.6）。

抢占版本规则：

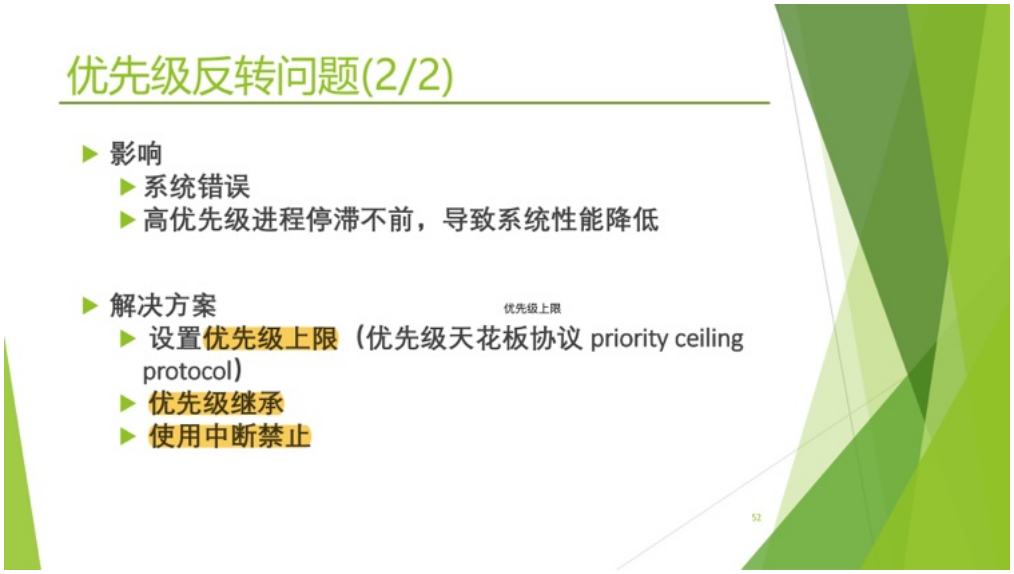
- 当有更高优先级的进程就绪时，可抢占 CPU；
- 被抢占的进程回到原级（实现上常放队首，避免与“因阻塞回到原级”的语义冲突）。

4.8.7 公平共享 / 保证调度 / 彩票调度

- **公平共享调度**：按用户或用户组分配 CPU 份额，已超份额的用户后续任务被压低优先级。



图示：低优先级 L 占有临界资源、被中等优先级 M 抢占，导致高优先级 H 等待 L 的资源、间接被 M 阻塞。

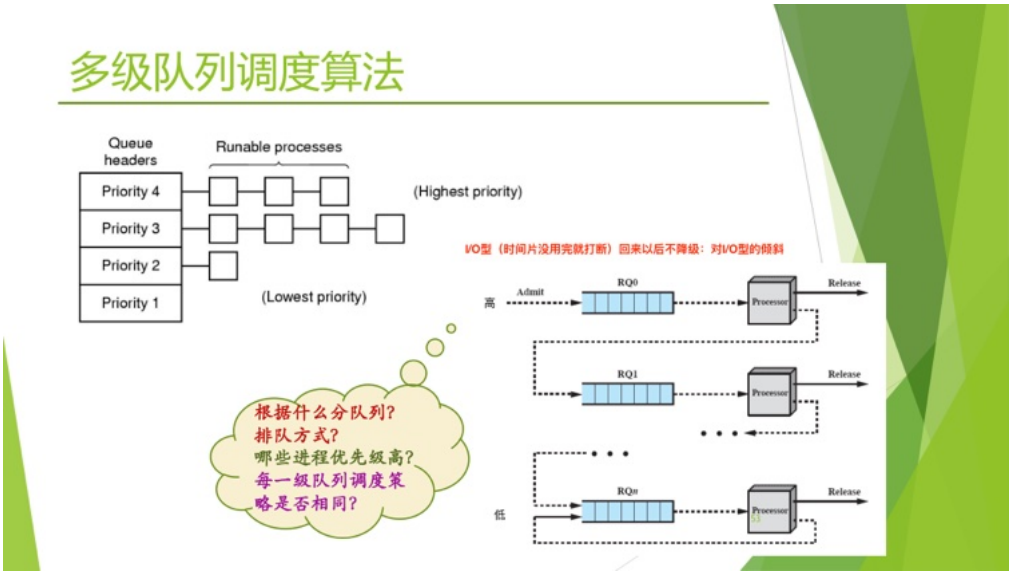


图示：可能导致系统错误或性能下降；解决方案包括优先级上限协议、优先级继承、关中断。

- **保证调度**：系统承诺给每个任务一个固定的 CPU 时间比例，监控实际使用并调整。
- **彩票调度（Lottery）**：每个任务持若干“彩票”，每次抽签决定下一个上 CPU 的任务；获得 CPU 的概率正比于持有彩票数。
- 公平共享思路是后来 Linux CFS 等公平调度器的来源。

4.8.8 调度算法对比表

记忆要点：

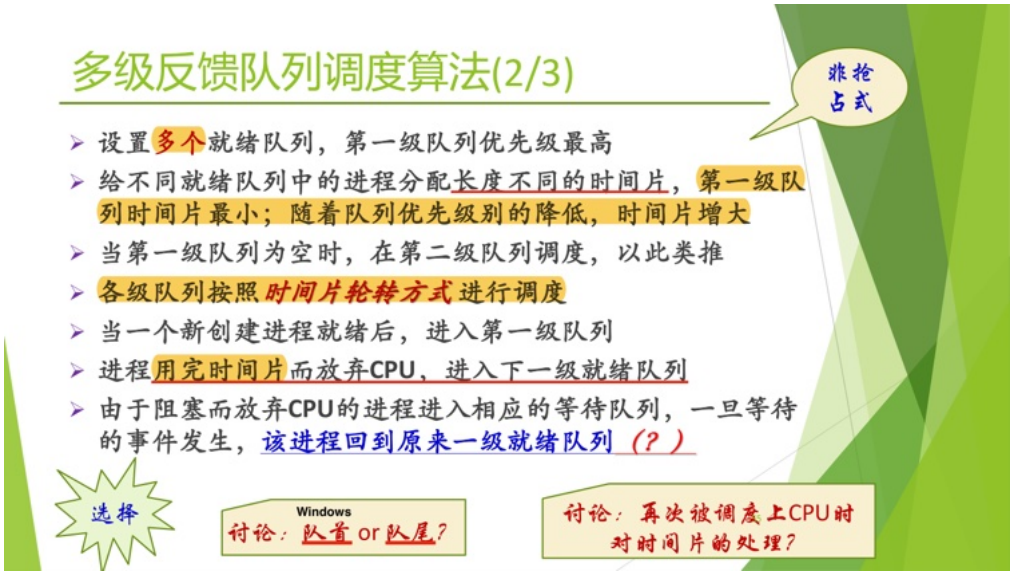


图示：每个就绪队列对应一类进程，通过优先级 + 各自调度算法组合。

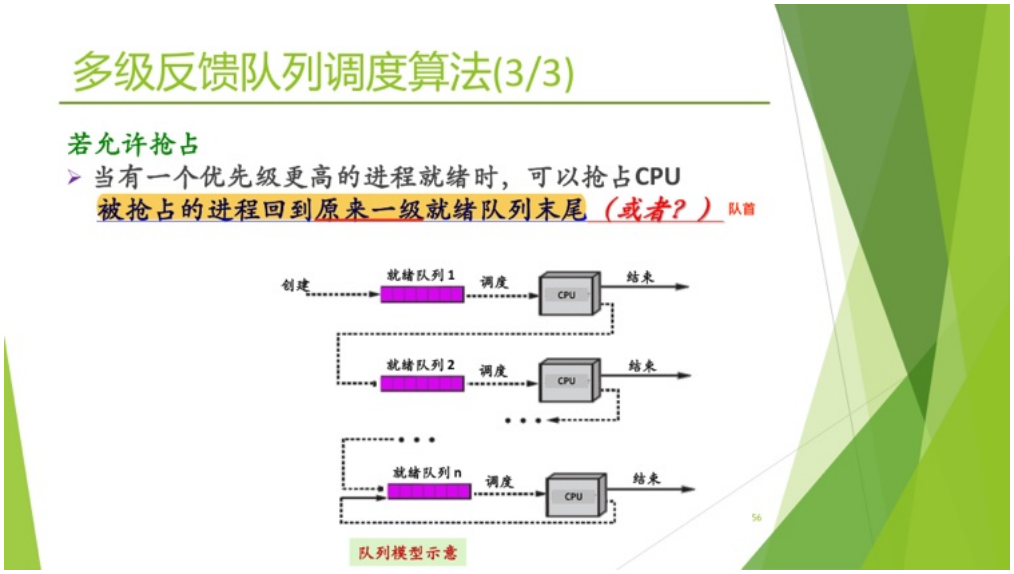


图示：1962 年由 Corbato 提出，CTSS 与 4.3 BSD 早期采用。

算法	选择函数	决策模式	吞吐量	响应	公平性	饥饿
FCFS	max[w]	非抢占	不强调	差	公平（按到达）	否
RR	常数	抢占（时间片）	时间片小则吞吐低	短任务响应好	公平	否
SJF	min[s]	非抢占	高	短任务好	不公平	可能
SRTN	min[s−e]	抢占（到达时）	高	提供好响应	不公平	可能
HRRN	max((w+s)/s)	非抢占	高	提供好响应	平衡	否
Feedback	见算法	抢占（时间片）	不强调	不强调	多目标	可能



图示：第 1 级最高优先级；时间片由小到大；新进程进 1 级；用完时间片下降；阻塞回原级。



图示：允许更高优先级的就绪进程抢占；被抢占的进程回到原级队尾或队首（设计选择）。

4.9 机制与策略分离

- 机制（mechanism）：上下文切换、PCB、就绪队列、原语、时钟中断；
- 策略（policy）：调度算法、时间片、优先级动态调整；
- 分离的好处：
  - 调度策略可以替换，不必动核心机制；
  - 不同应用 / 用户可以选择不同策略。

后续 §10、§11 中“线程调度器是机制不变 + 策略灵活”的思路就是这条原则的实例。



图示：公平共享（Fair-share）、保证调度（Guaranteed）、彩票调度（Lottery）。



图示：FCFS / RR / SJF / SRTN / HRRN / Feedback 在选择函数、决策模式、吞吐量、响应时间、开销、对进程影响、饥饿问题上的对比。

4.10 典型操作系统的调度算法

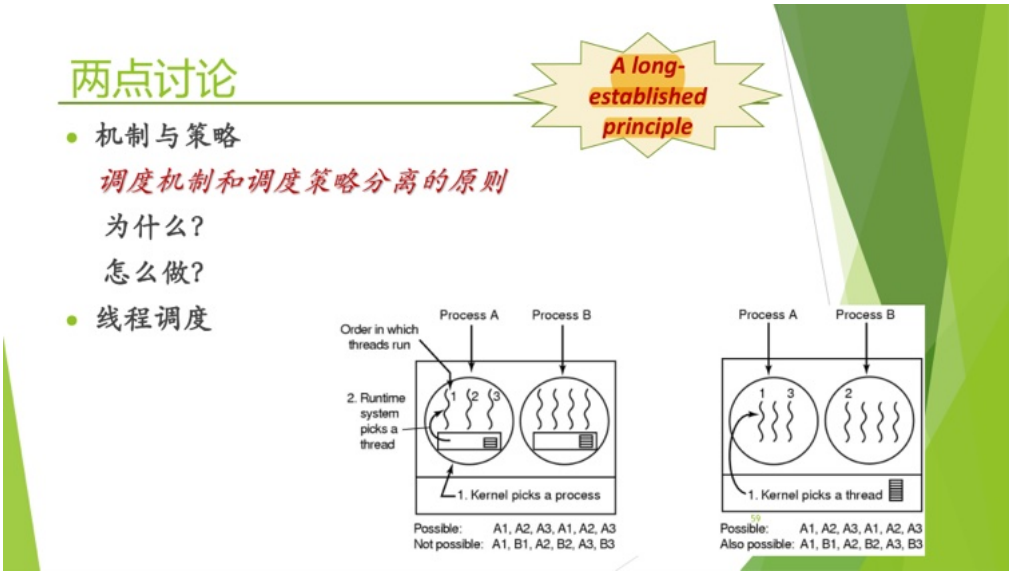
下面 §10、§11 展开 Windows 与 Linux 的细节。

4.11 Windows 线程调度

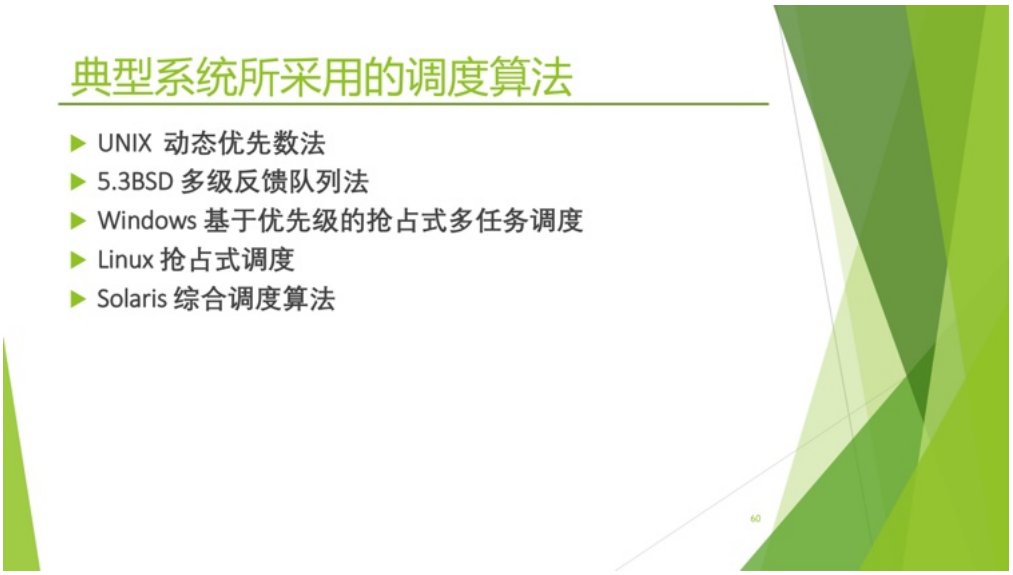
4.11.1 总体特征

要点：

- 调度单位是线程，多个就绪队列按优先级排队；
- 总是选择最高优先级的就绪线程运行；



图示：A long-established principle ——为什么、怎么做。

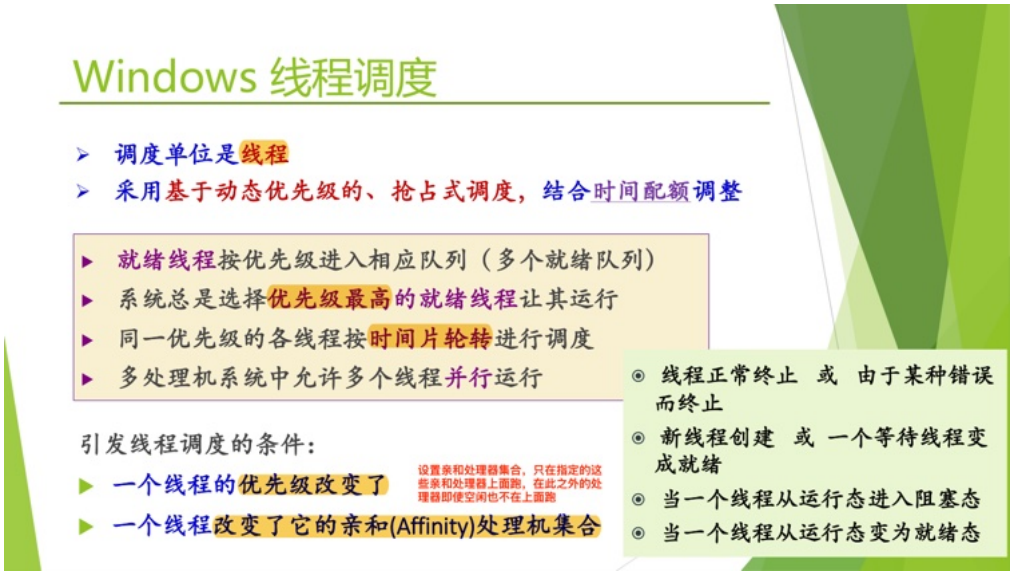


图示：UNIX 动态优先数法 / 4.3 BSD 多级反馈 / Windows 抢占式优先级 / Linux 抢占式 / Solaris 综合调度。

- 同优先级按 RR 调度；
- 多 CPU 上允许多个线程并行运行。

引发线程调度的事件：

- 线程优先级改变；
- 线程改变其亲和（Affinity）处理机集合；
- 线程正常 / 异常终止；
- 新线程创建、等待线程变就绪；
- 线程从运行态进入阻塞态、运行态变就绪态等。



图示：调度单位是线程，基于动态优先级、抢占式、结合时间配额调整。

> **亲和处理器集合**：线程被设置只在该集合中的处理器上运行；集合外即使空闲也不上。

4.11.2 线程优先级

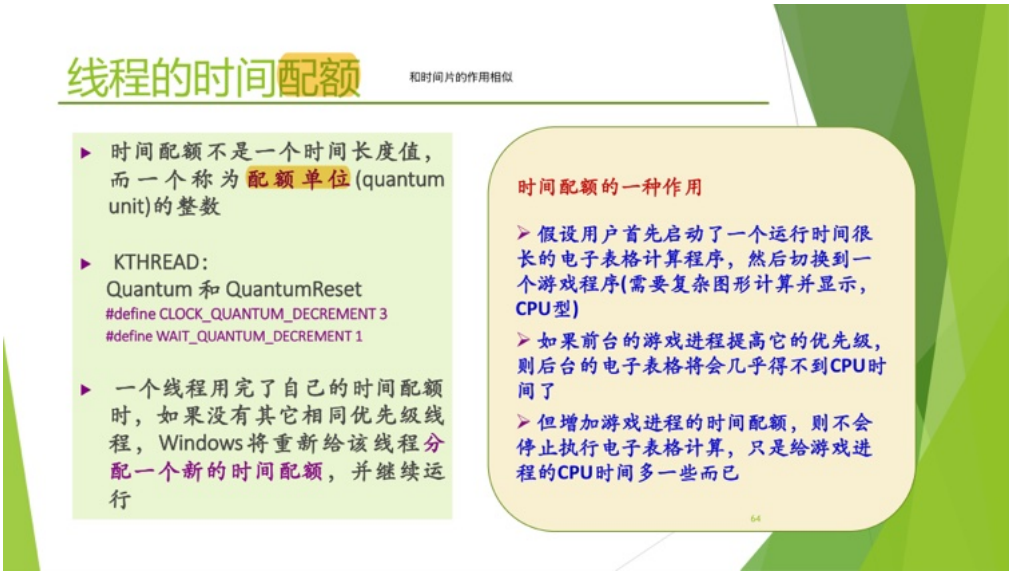


图示：0 是零页线程；1–15 可变优先级；16–31 实时优先级；系统线程在更高内部位置。

- **零页线程**：用于把空闲物理页面清零；
- **实时优先级线程**：优先级不会被改；
- **可变优先级线程**：在可变区间内能被升降；
- 区分**基本优先级**和**当前优先级**（可被提升）。

4.11.3 时间配额

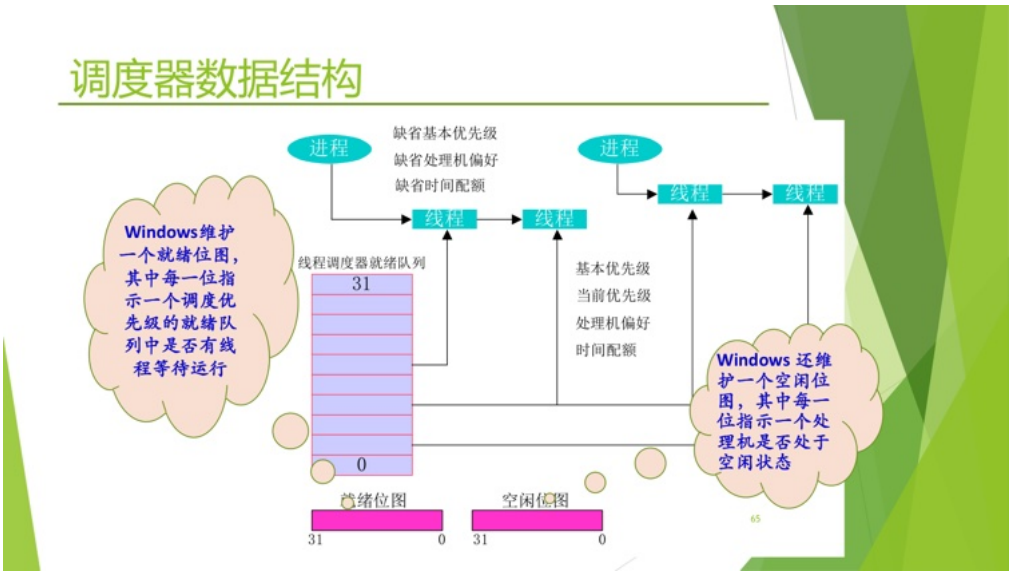
- 时间配额不是“多少毫秒”，而是用配额单位（quantum unit）整数表示；



图示：配额单位（quantum unit）是整数；KTHREAD::Quantum, QuantumReset。

- 时钟中断到来减 3，等待结束后减 1（`CLOCK_QUANTUM_DECREMENT = 3`，`WAIT_QUANTUM_DECREMENT = 1`）；
- 配额用完后：
  - 同优先级有其他就绪线程：切换；
  - 否则给该线程重新分配新配额继续运行；
- 时间配额的一个用法：把游戏进程的时间配额加大，让它得到更多 CPU，而不是简单提优先级；这样不会饿死后台电子表格。

4.11.4 调度器数据结构

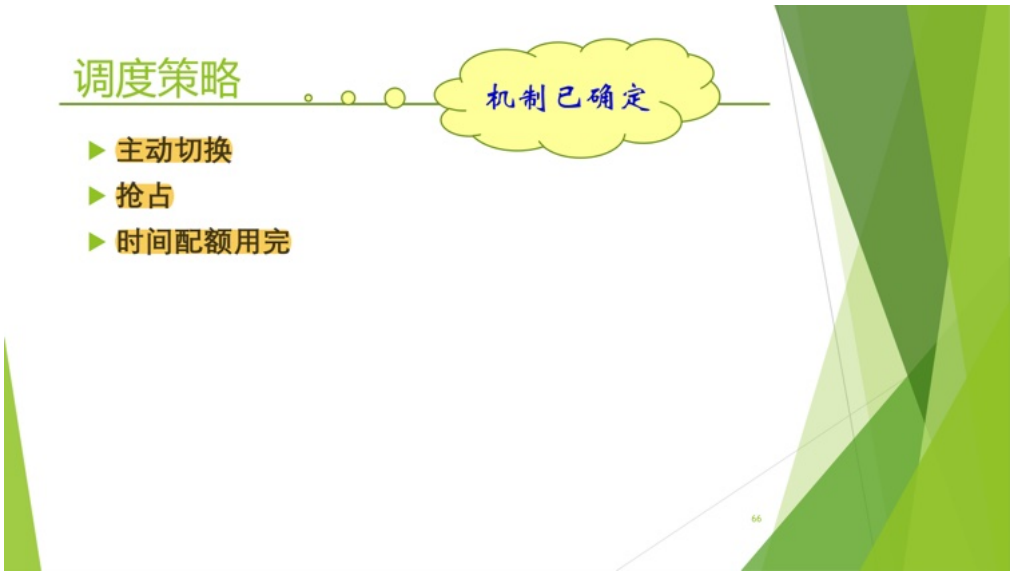


图示：32 位的就绪位图、32 位的空闲位图、就绪队列、亲和与基本/当前优先级。

- **就绪位图**：每位指示一个调度优先级的就绪队列是否非空；
- **空闲位图**：每位指示一个处理器是否空闲；

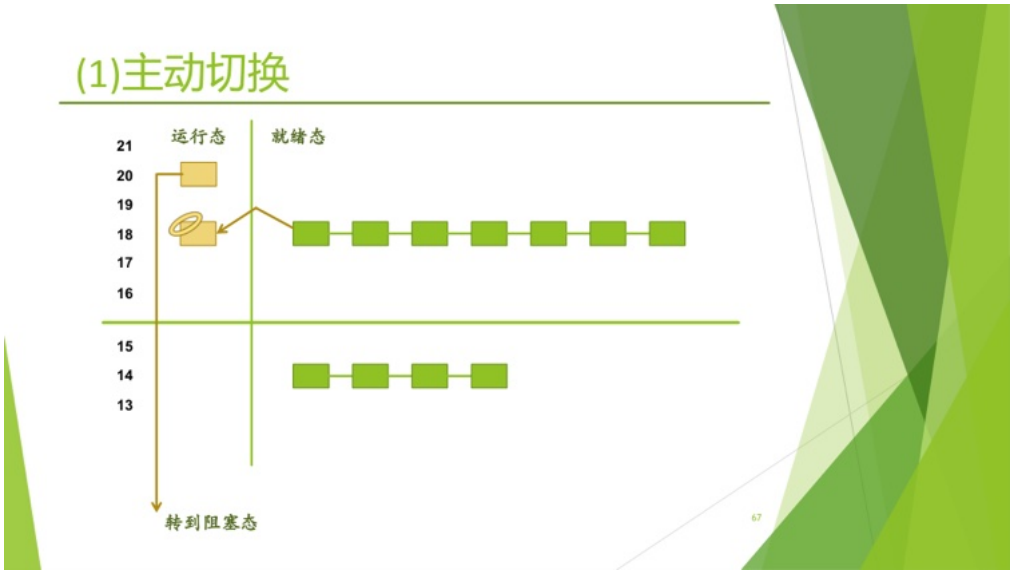
- 这两个位图加上就绪队列 / 进程链表是调度器“快速选下一个线程”的数据基础。

4.11.5 调度策略：主动切换 / 抢占 / 配额用完



图示：机制确定后，策略由三种触发方式构成。

主动切换

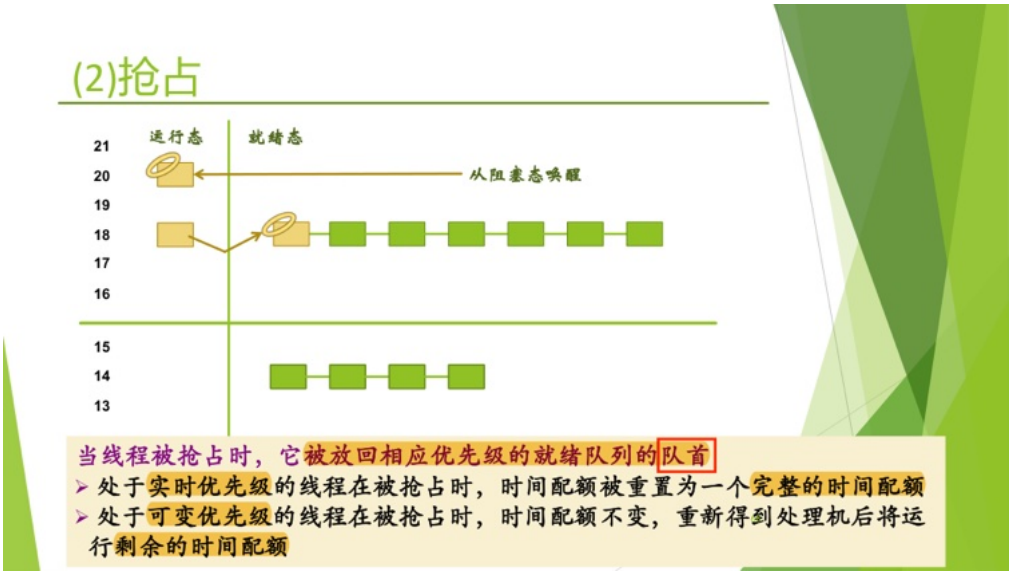


图示：线程主动让出（Sleep、WaitFor*）-> 转到阻塞态。

- 线程主动调用阻塞函数；
- 把当前线程放入相应的等待队列；
- 选取就绪位图中最高优先级队列的线程运行。

抢占

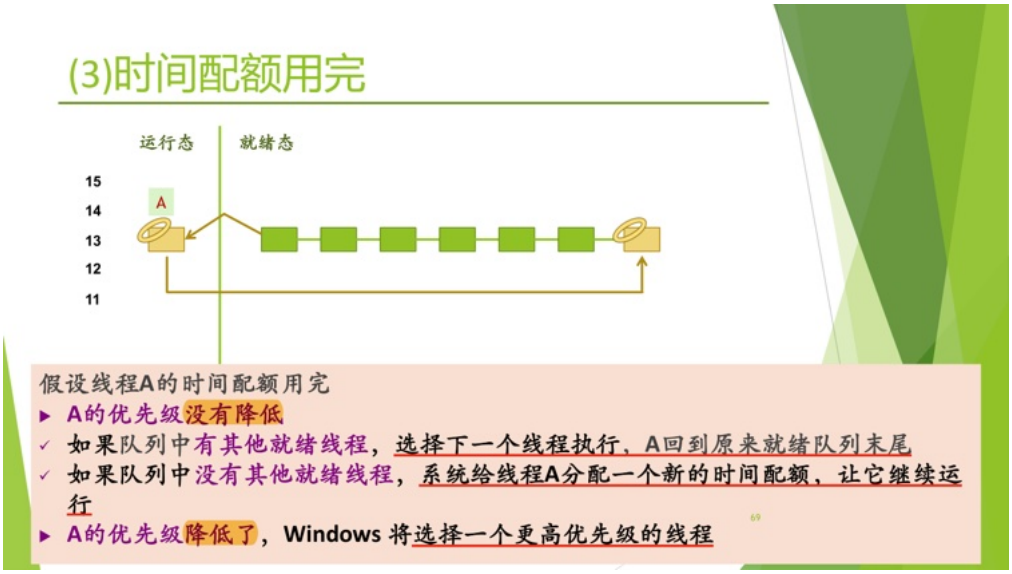
- 实时优先级线程被抢占时，时间配额被重置为完整配额；



图示：线程被更高优先级线程抢占；放回相应优先级就绪队列队首。

- 可变优先级线程被抢占时，时间配额不变，重新得 CPU 后运行剩余配额。

时间配额用完

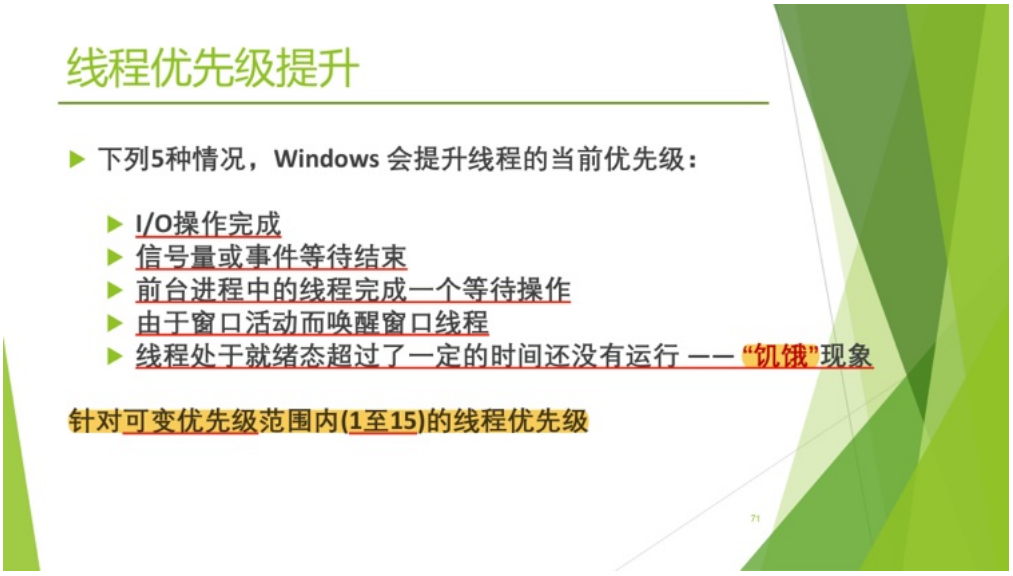


图示：配额用完后的两种情况。

- 假设线程 A 配额用完：
  - 优先级未降低：
    - 队列还有其他就绪线程 -> 选下一个执行，A 进队尾；
    - 队列没有其他就绪线程 -> 给 A 一个新配额，继续运行；
  - 优先级降低了 -> 选更高优先级线程运行。

4.11.6 线程优先级提升与时间配额调整

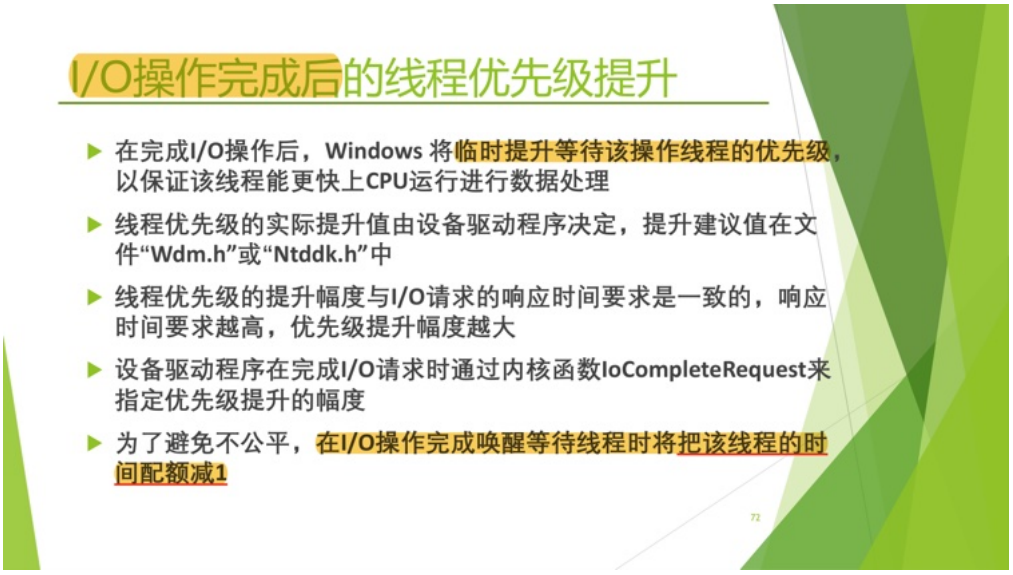
针对可变优先级（1–15）的线程，下面五种情形会被提升：



图示：5 种情形下 Windows 会临时提升线程优先级。

- I/O 操作完成；
- 信号量或事件等待结束；
- 前台进程线程完成等待；
- 窗口活动唤醒窗口线程；
- 处于就绪态超过一定时间还没运行（饥饿）。

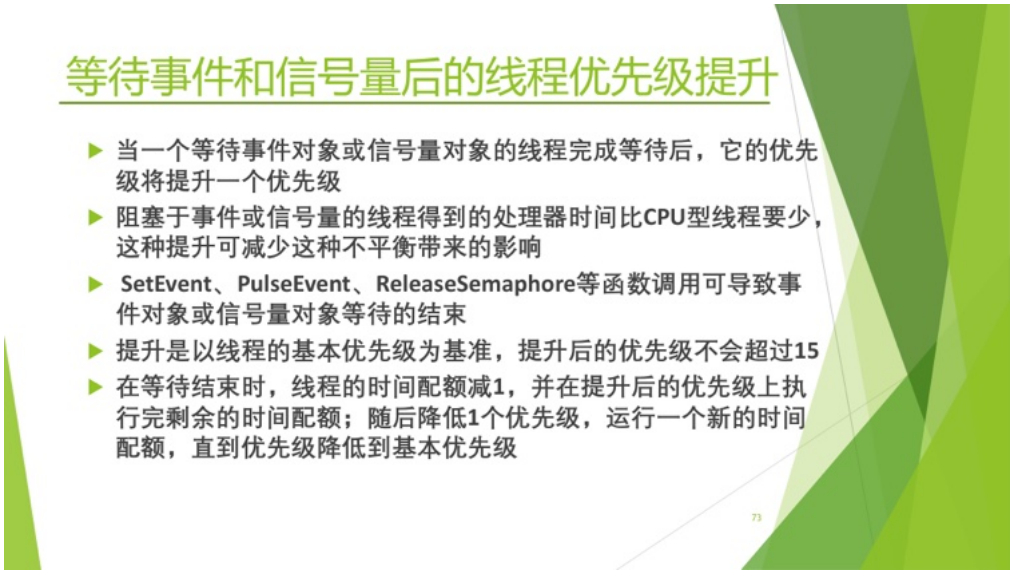
I/O 完成后的提升



图示：临时提升优先级，按设备驱动建议值；唤醒时把时间配额减 1。

- 提升幅度由设备驱动决定，建议值在 `Wdm.h` 或 `Ntddk.h`；
- 响应时间要求越高，提升幅度越大；
- 防止不公平：唤醒时配额减 1。

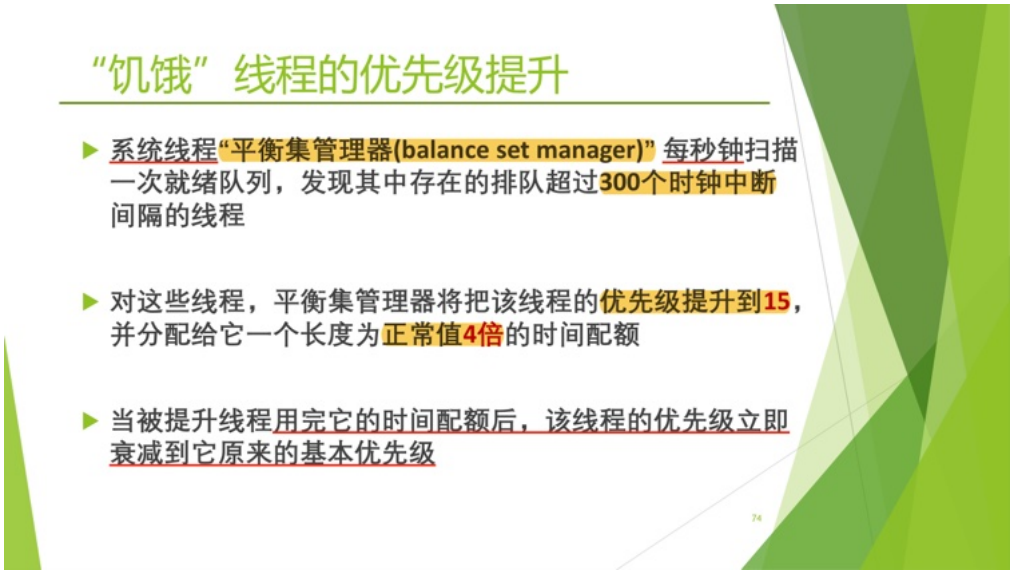
等待事件 / 信号量后的提升



图示：提升一个优先级，最高不超过 15；用完后逐级衰减回基本优先级。

- 用以补偿等待这些对象的线程获得的处理器时间不足；
- 提升以基本优先级为基准；
- 配额减 1，运行结束后逐级降回基本优先级。

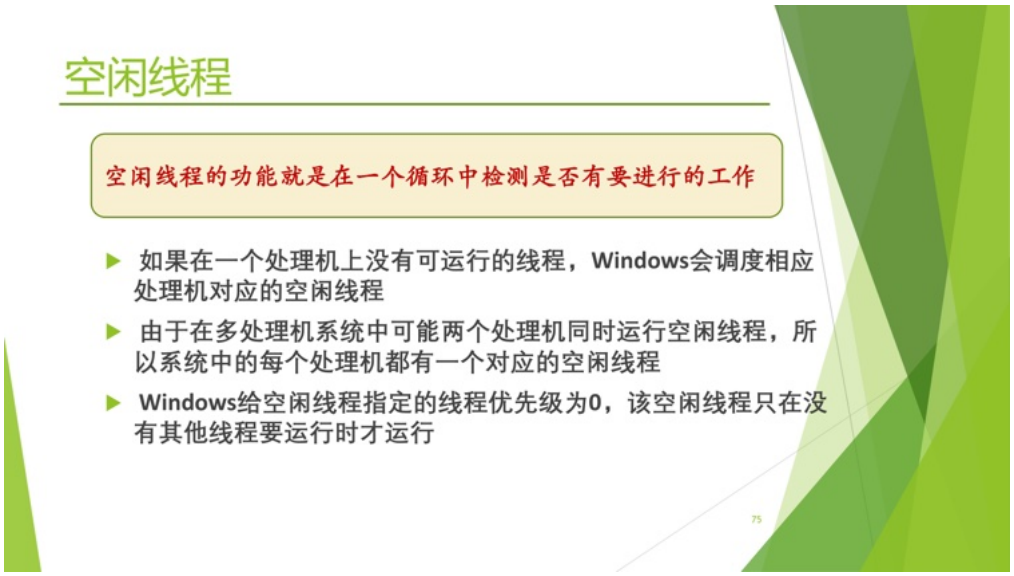
“饥饿”线程的提升



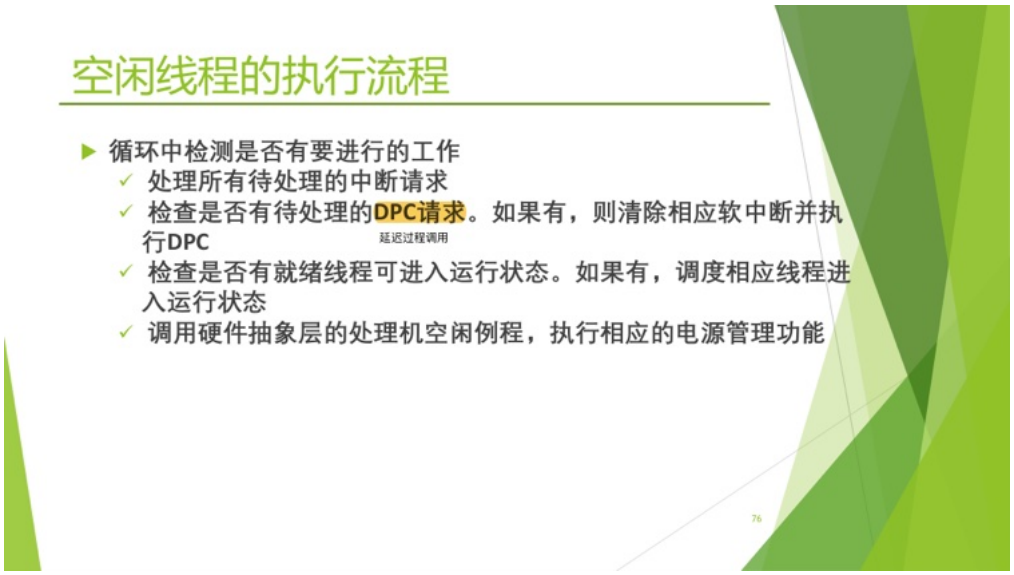
图示：平衡集管理器（balance set manager）每秒扫描就绪队列。

- 系统线程“**平衡集管理器**”每秒扫描一次就绪队列；
- 发现排队超过 300 个时钟中断间隔的线程，把它提升到 15 并分配 4 倍正常时间配额；
- 配额用完后立即降回基本优先级。

4.11.7 空闲线程



图示：每个处理机有一个空闲线程，优先级 0；只在没其他线程要运行时才运行。



图示：处理待处理中断、检查 DPC 请求、检查就绪线程、调用电源管理空闲例程。

- 多处理机系统中每个处理机都有一个空闲线程；
- 在循环中：
  1. 处理所有待处理的中断请求；
  2. 检查并执行待处理的 DPC（延迟过程调用）；
  3. 检查是否有就绪线程，若有就调度；
  4. 调用 HAL 处理机空闲例程，执行电源管理功能。

4.12 多处理器调度

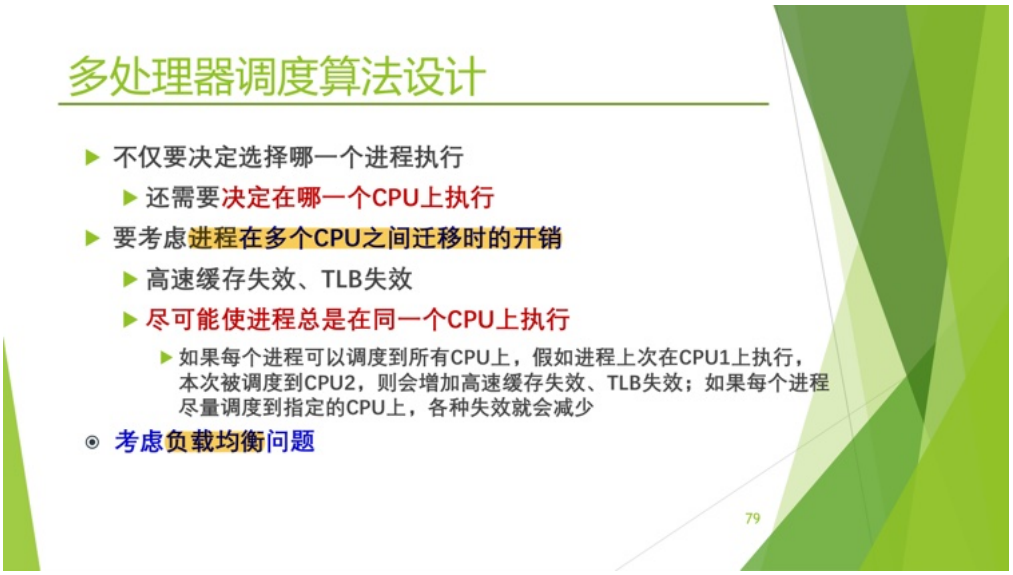
4.12.1 多处理器调度的特征



图示：SMP 系统每个处理器运行自己的调度程序；处理器间负载共享；共享资源访问需要同步（加互斥锁）。

- 多处理器组成多处理机系统；
- SMP 模型中每个处理器有自己的调度程序；
- 处理器间可负载共享；
- 访问共享资源（如全局就绪队列）需要锁。

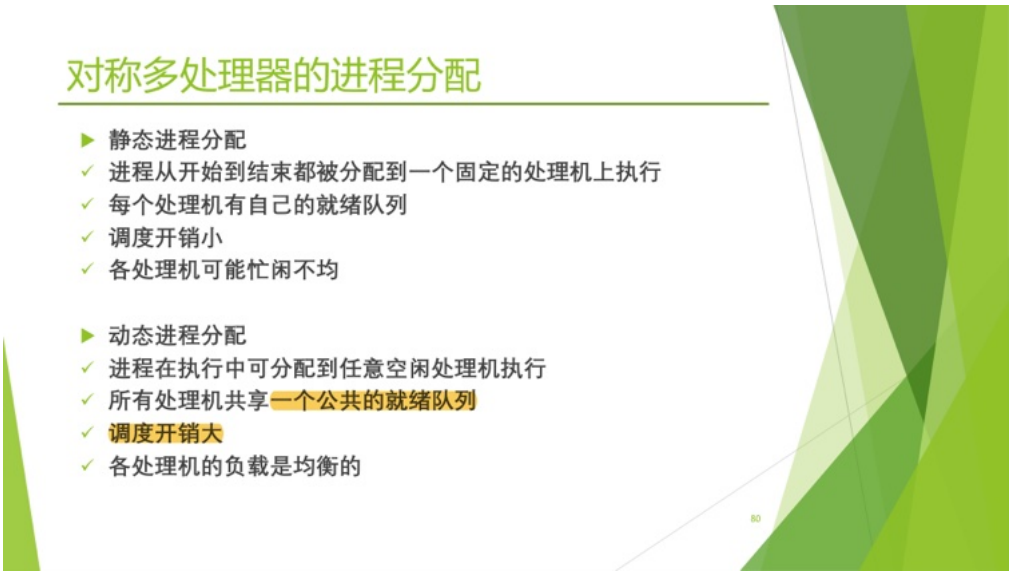
4.12.2 多处理器调度算法的设计



图示：不仅决定选哪个进程，还要决定在哪一个 CPU 上执行；要考虑迁移开销与负载均衡。

- 进程在不同 CPU 间迁移会带来 cache、TLB 失效；
- 尽量让进程在同一 CPU 上执行（缓存亲和性）；
- 同时要在负载均衡和缓存亲和性间折中。

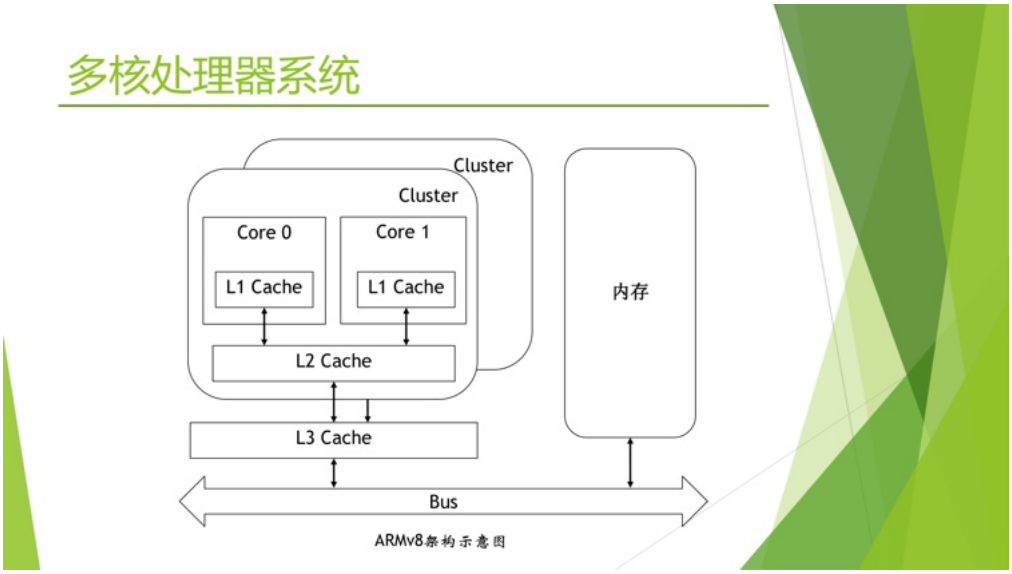
4.12.3 静态 / 动态进程分配



图示：静态分配开销小但易忙闲不均；动态分配负载均衡但开销大。

- **静态分配**：进程从开始到结束固定在某 CPU；每 CPU 自己的就绪队列；调度开销小；但负载可能不均。
- **动态分配**：进程在执行中可调度到任意空闲 CPU；所有处理机共享公共就绪队列；负载均衡但开销大。

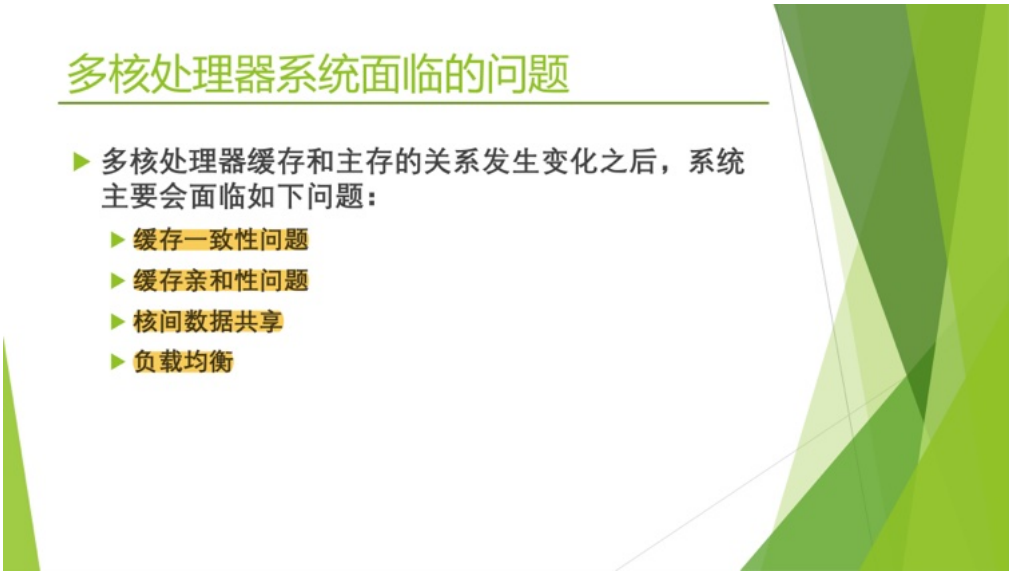
4.12.4 多核处理器的特殊问题



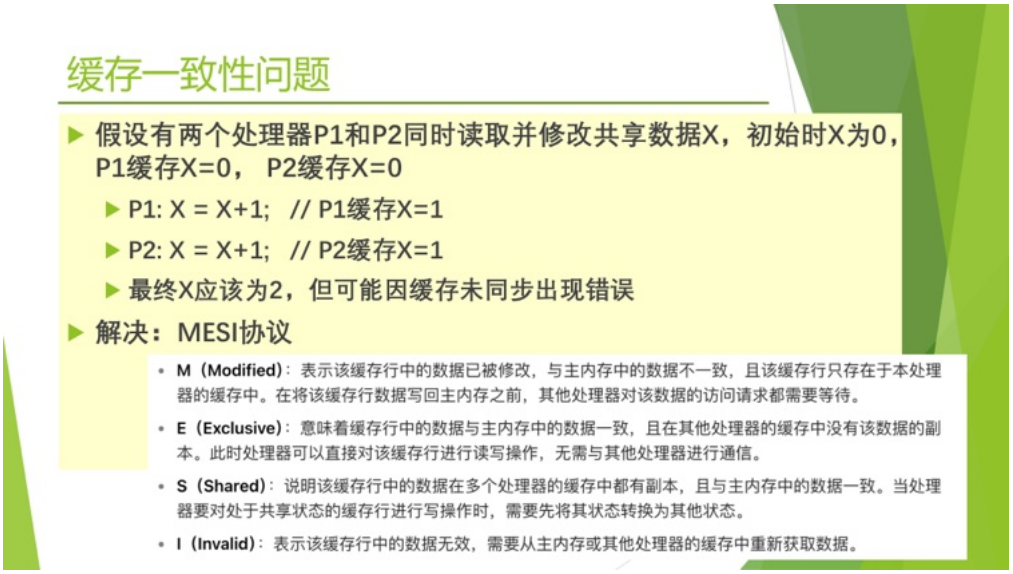
图示：ARMv8 架构示意——L1/L2/L3 cache、cluster、内存。

缓存一致性

- 多核共享主存，但每核有私有 L1/L2 缓存；
- 同一数据在多个核 cache 中可能有不同副本；



图示：缓存一致性、缓存亲和性、核间数据共享、负载均衡。



图示：两个核同时修改共享数据 X，可能造成一致性问题；解决方案是 MESI 协议。

- MESI（Modified / Exclusive / Shared / Invalid）协议保证一致性；
- 这是后面同步原语（spinlock、原子操作）的硬件基础。

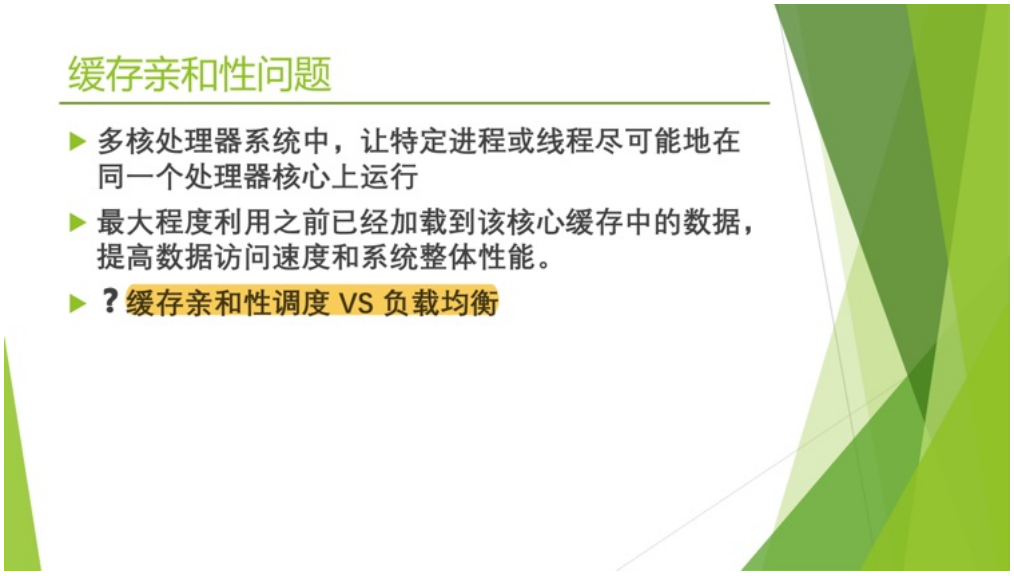
缓存亲和性

- 让特定进程或线程尽可能跑在同一核心上，复用之前的 cache；
- 与负载均衡常常冲突，需要权衡。

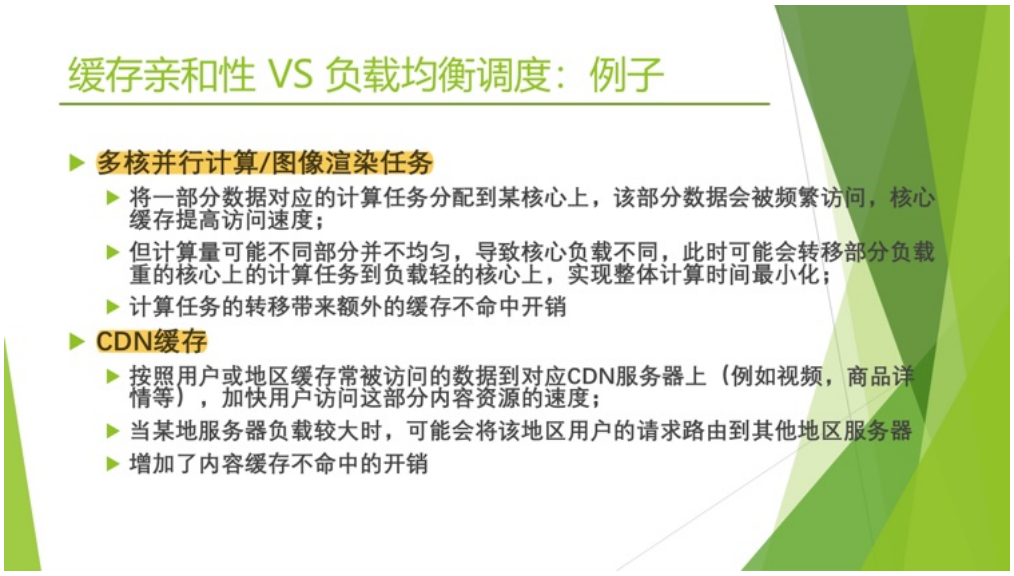
负载均衡：作业偷取（Work Stealing）

核心思想：

- 任务按策略分配到各处理器的本地队列；
- 优先执行本地任务；



图示：让进程 / 线程尽量在同一核上运行，复用缓存中的数据。



图示：多核渲染、CDN 缓存等场景中，缓存亲和性与负载均衡如何权衡。

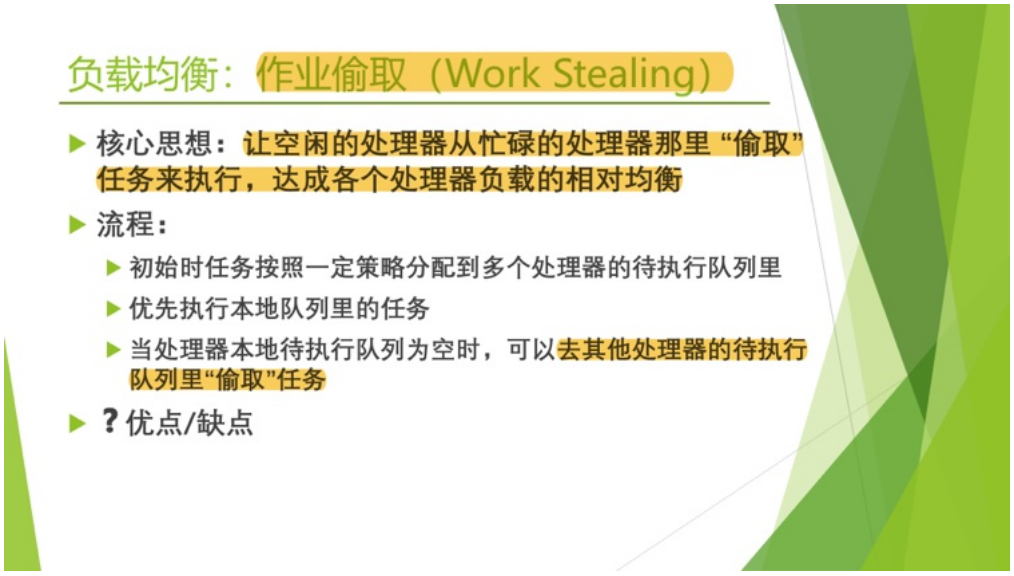
- 本地空了就去别人的队列“偷”任务执行；
- 优点：去中心化、扩展性好；缺点：跨核访问 cache miss 多。

案例：Go GMP

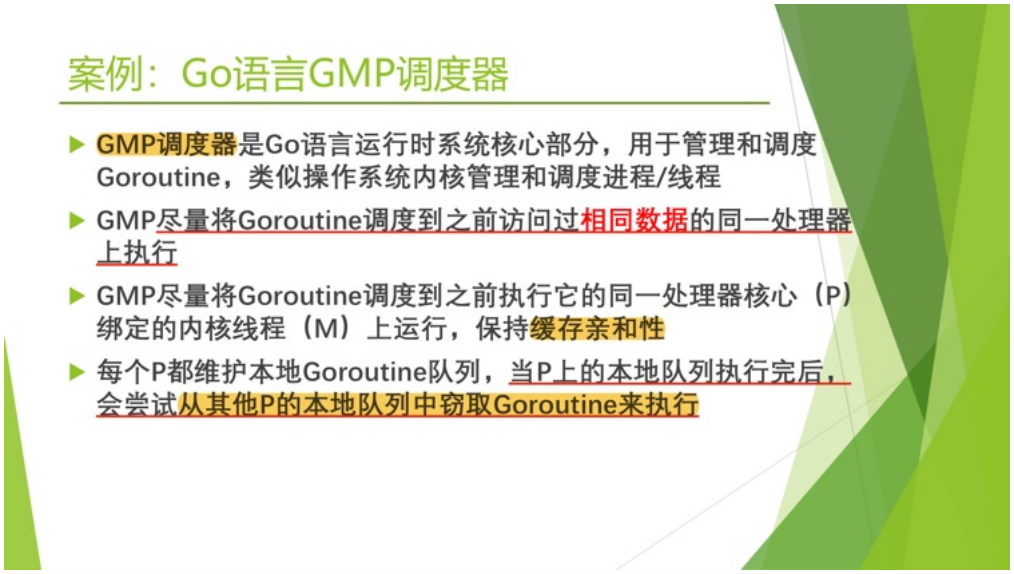
- GMP 把 Goroutine 调度到曾经访问相同数据的同一处理器；
- 把 Goroutine 调到上次执行它的同一 P 绑定的内核线程 M 上；
- 每个 P 有本地 Goroutine 队列；本地为空时尝试从其他 P 偷任务。

案例：YARN

- 分层：资源管理层（NodeManager）+ 应用调度层（ResourceManager）；
- NodeManager 管节点上的资源、容器；



图示：空闲处理器从忙碌处理器的任务队列偷任务。



图示：Go 运行时把 Goroutine 映射到内核线程，结合 P（Processor）维持本地队列与缓存亲和性。

- 利用 HDFS 等分布式文件系统的位置信息把任务分给数据所在节点；
- 支持多队列调度，不同队列设置不同分配策略。

4.13 实时调度

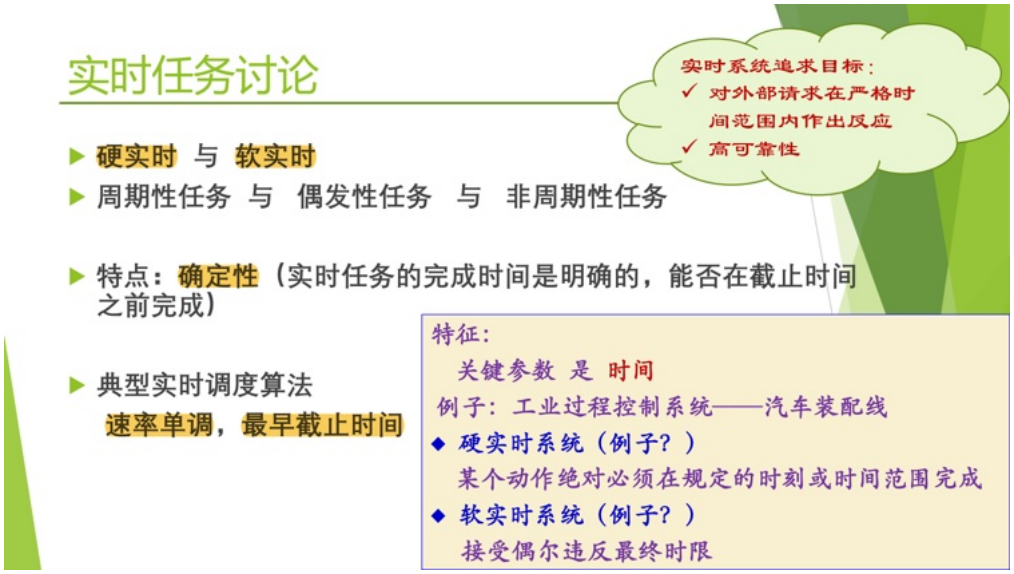
- **硬实时（Hard Real-time）**：必须在严格时间内完成，例如汽车装配线、飞行控制；
- **软实时（Soft Real-time）**：偶尔违反最终时限可接受，例如视频播放；
- 任务类型：周期性、偶发性、非周期性。

4.13.1 可调度性

可调度性条件：



图示：分层资源管理 + 应用调度；NodeManager 管节点资源，ResourceManager 协调全局。



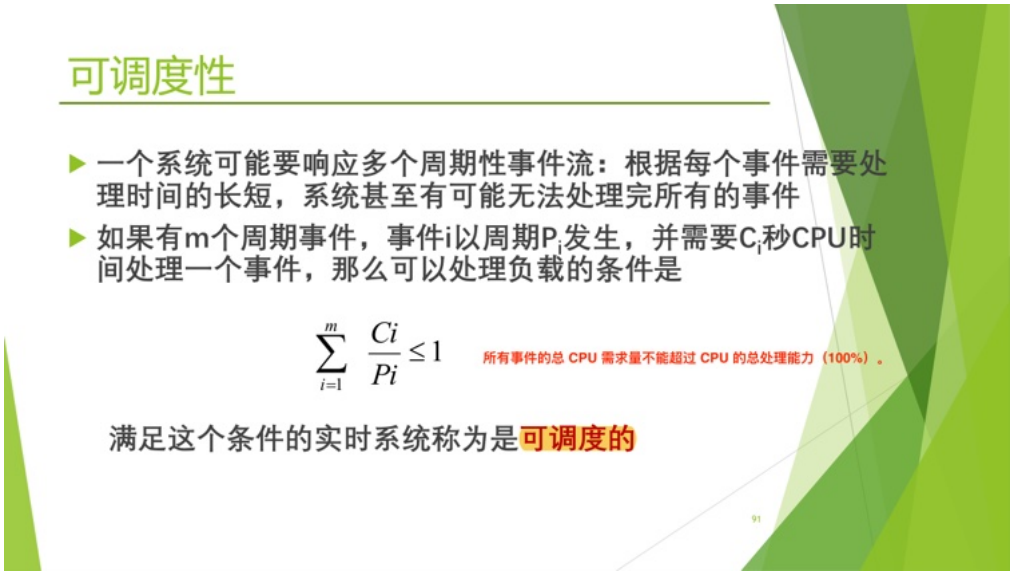
图示：硬实时 vs 软实时；周期 / 偶发 / 非周期任务；典型实时调度算法 RM 与 EDF。

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

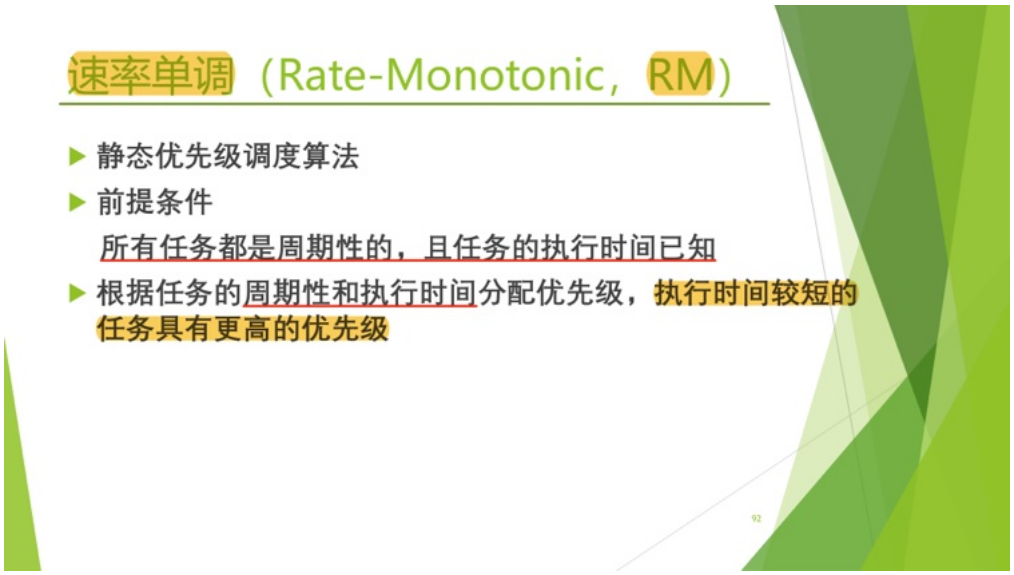
含义：所有周期任务的总 CPU 需求量不能超过 100%。

4.13.2 速率单调 RM

- 静态优先级、抢占式；
- 周期越短优先级越高；
- 简单、易实现；
- 静态优先级算法中的最优算法之一；
- 缺点：只适用于周期性任务；CPU 利用率上限不如动态优先级算法。



图示：m 个周期事件，事件 i 周期 P_i、CPU 时间 C_i，可调度条件 (C_i / P_i) <= 1。



图示：周期短的任务静态优先级高；要求所有任务都是周期性、执行时间已知。

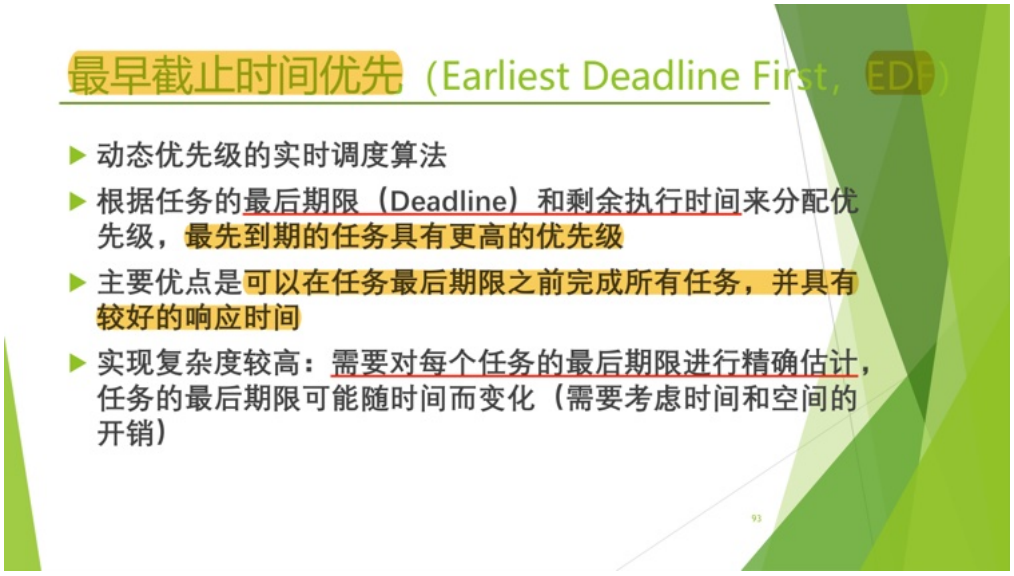
4.13.3 最早截止时间优先 EDF

- 动态优先级、抢占式；
- 截止时间最近的任务优先；
- 在抢占、独立任务的理想模型下，只要总利用率 <= 100%，EDF 都能调度；
- 缺点：实现复杂；多米诺骨牌效应（一个任务错过最后期限，后续任务可能连锁错过）。

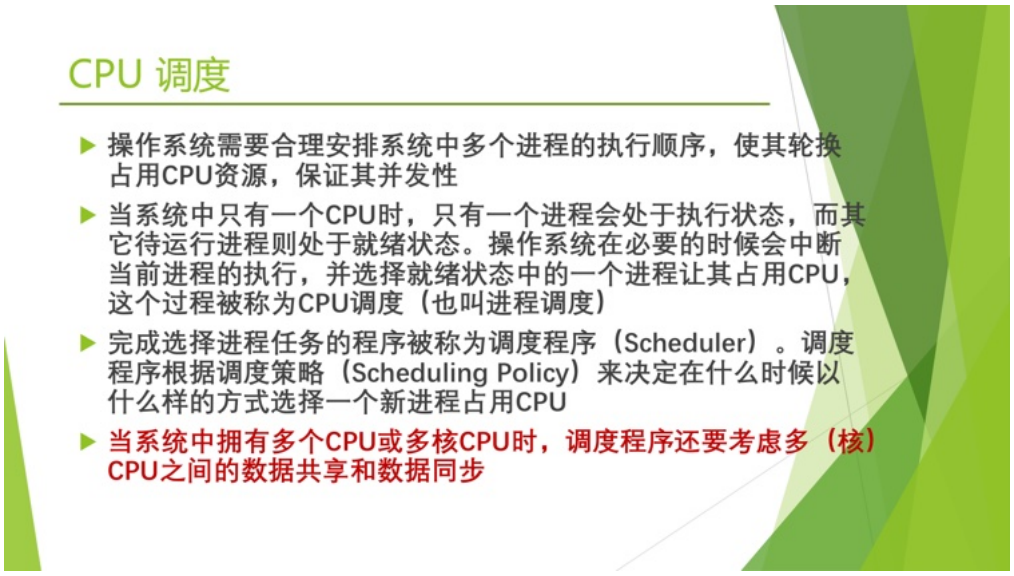
4.14 openEuler 的多核调度

4.14.1 CPU 调度概念

- 调度程序根据策略决定何时、用什么方式选择新进程占用 CPU；



图示：动态优先级，截止期限越近优先级越高；利用率可达 100%；但实现复杂。



图示：从单 CPU 到多核 CPU，调度程序需要考虑数据共享与同步。

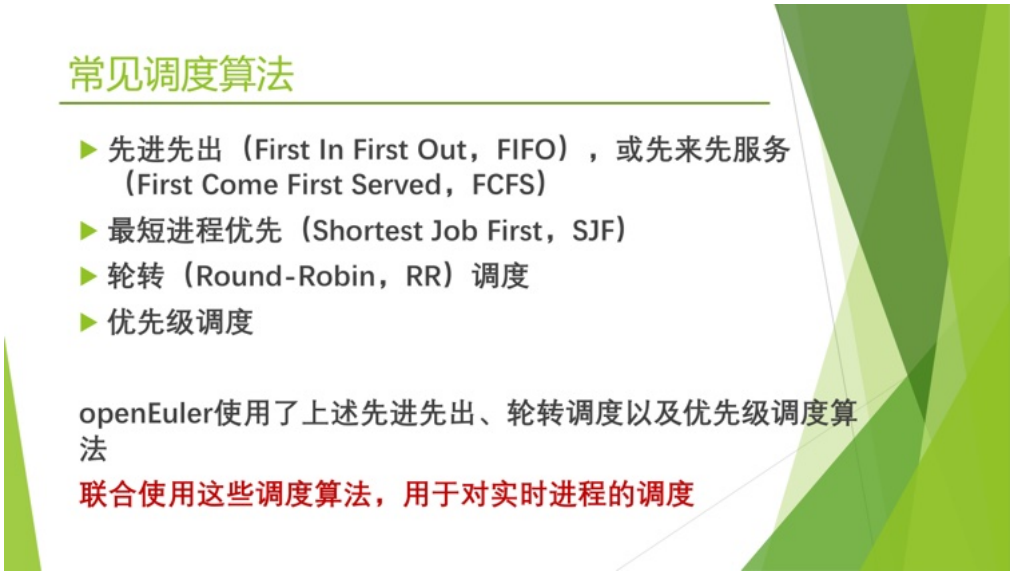
- 多 CPU 时还要考虑多核之间的数据共享和同步。

4.14.2 常见调度算法

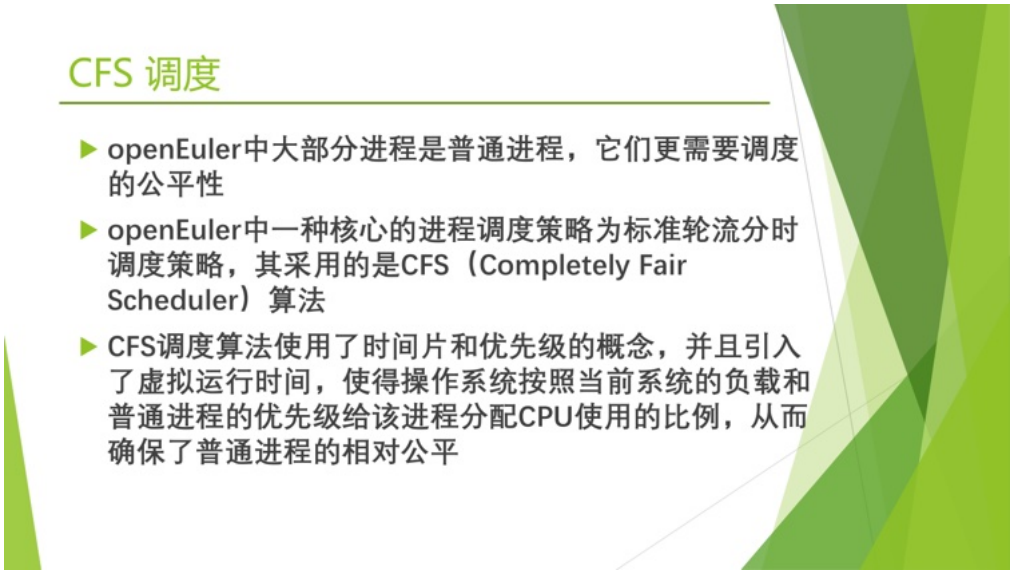
4.14.3 CFS 调度

- CFS 使用时间片和优先级的概念；
- 引入虚拟运行时间（vruntime）；
- 按当前系统负载和优先级分配 CPU 比例，确保普通进程的相对公平。

4.14.4 多核处理器系统



图示：FIFO/FCFS、SJF、RR、优先级；openEuler 联合使用 FIFO + RR + 优先级用于实时进程。



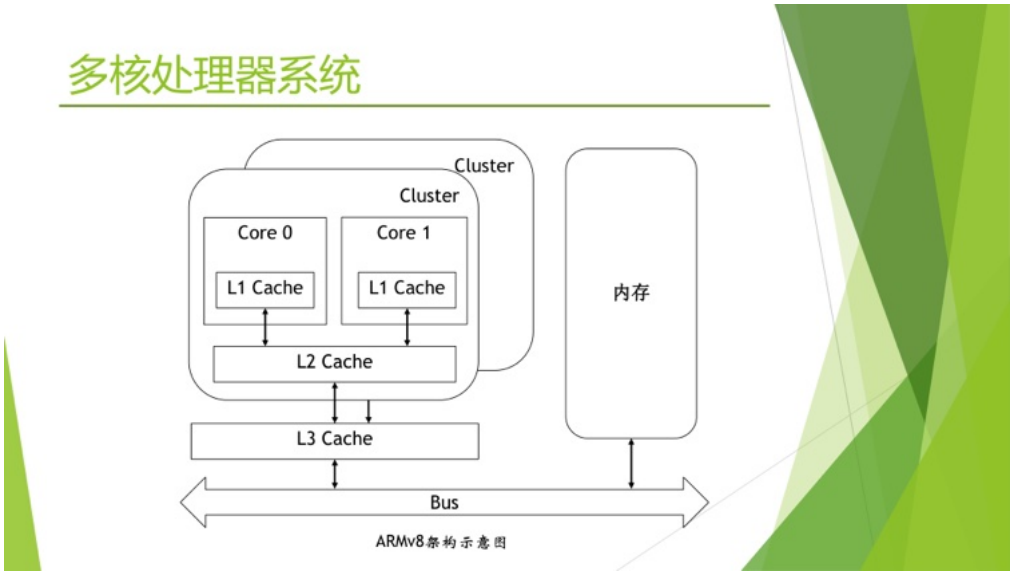
图示：openEuler 中普通进程使用标准轮流分时调度策略——CFS。

4.14.5 单队列 vs 多队列调度

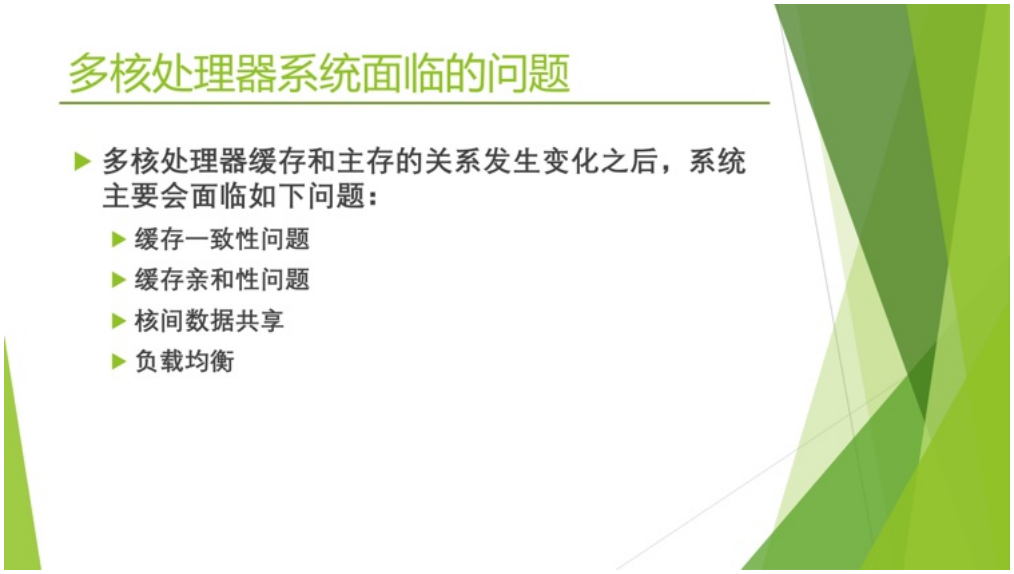
迁移线程要点：

- 每个 CPU 有一个迁移线程；
- 工作队列里挂的是要执行的迁移函数；
- 迁移线程不被其他进程抢占；
- 通过把任务从一个 CPU 移到另一个 CPU 实现负载均衡。

4.15 Linux 调度算法的演化



图示：ARMv8 多核 + L1/L2/L3 缓存。

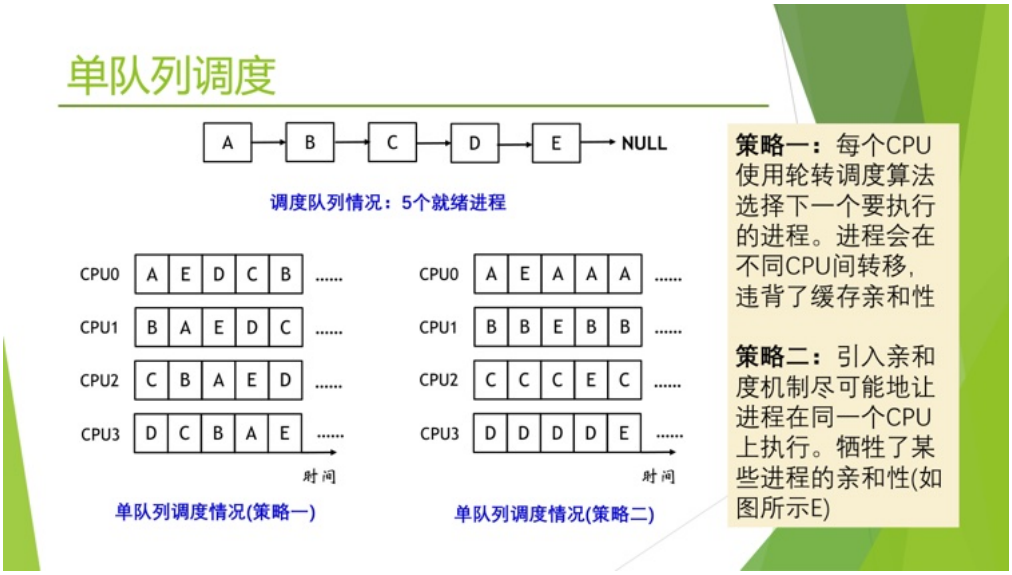


图示：缓存一致性、亲和性、核间共享、负载均衡。

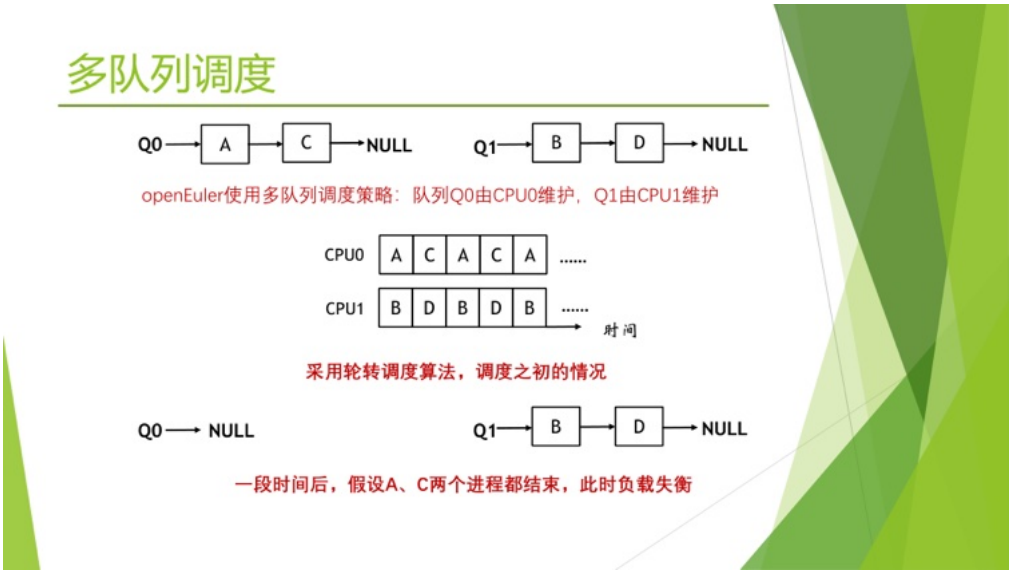
4.15.1 调度器演化历史

主要历程：

1. Linux 2.4 简单基于优先级调度；
2. Linux 2.6 O(1) 调度器（Ingo Molnar）；
3. Con Kolivas 的 SD（楼梯调度）补丁；
4. RSDL（旋转楼梯最后期限调度）补丁；
5. Linux 2.6.23 起的 CFS 完全公平调度；
6. Linux 6.11 起的 EEVDF。



图示：策略 1（每 CPU 用 RR 选下一个）破坏亲和性；策略 2 引入亲和度机制。



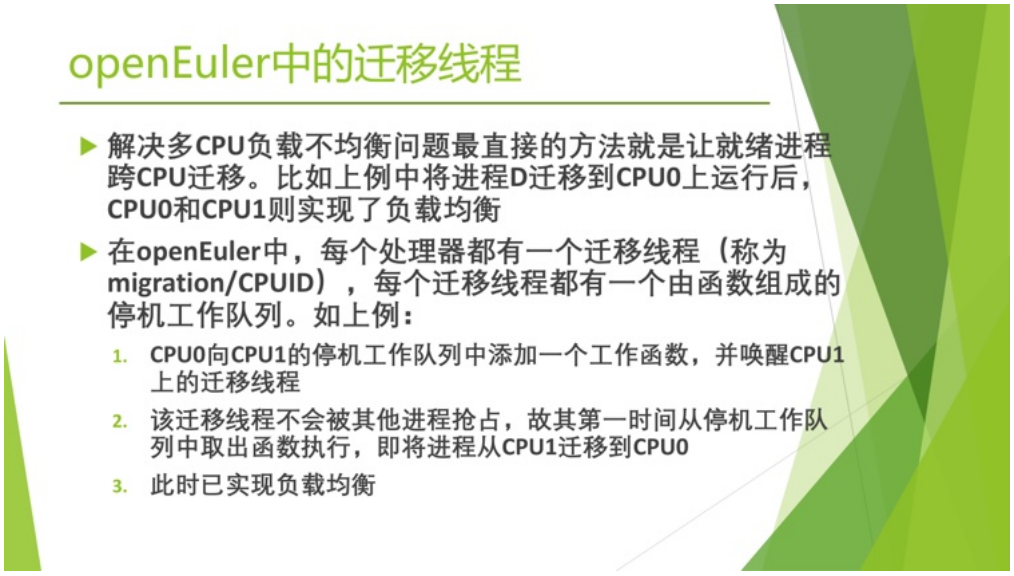
图示：openEuler 使用多队列调度：每个 CPU 维护自己的队列；任务结束后可能负载失衡。

4.15.2 Linux 进程分类

- 实时进程：调度延迟要求最高；调度策略 `SCHED_FIFO`、`SCHED_RR`；
- 普通进程：交互式进程对响应敏感、批处理进程对响应不敏感；都使用 CFS。

4.15.3 Linux 2.4 的调度器

- 普通进程优先级 = `counter + nice`；
- counter 越大优先级越高；
- 当所有 RUNNING 进程的 counter 用完，重算所有进程（包括睡眠的）的 counter；
- 睡眠进程 counter 没用完，重算时会加上未用完的部分 -> 提高其优先级；
- 这是 2.4 调度器对交互进程的补偿；



图示：每个 CPU 都有一个迁移线程 `migration/CPUID`，用停机工作队列搬迁任务。



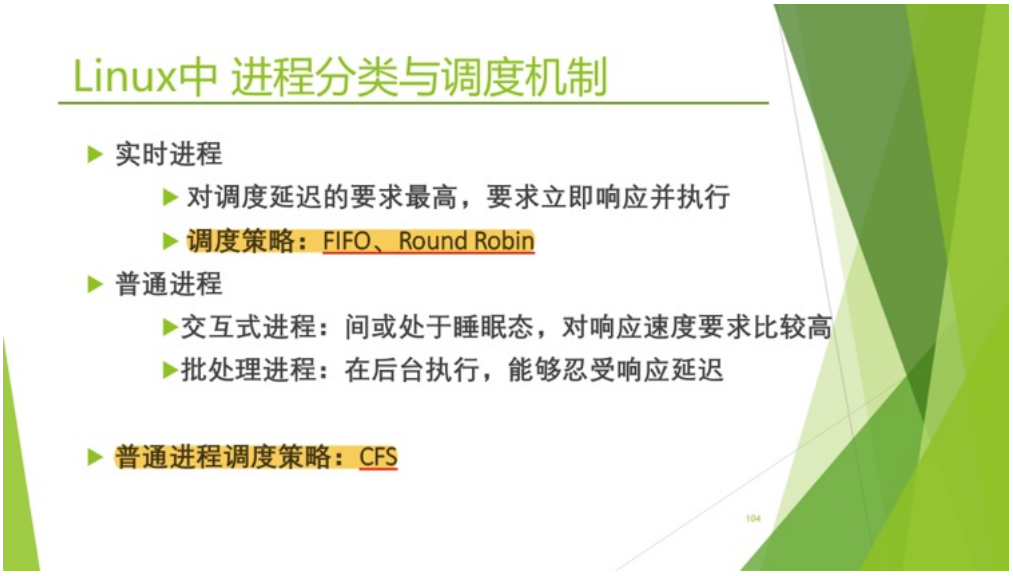
图示：2.4 简单优先级 -> 2.6 O(1) -> SD 补丁 -> RSDL 补丁 -> CFS -> 6.11 EEVDF。

- 2.4 内核是非抢占的。

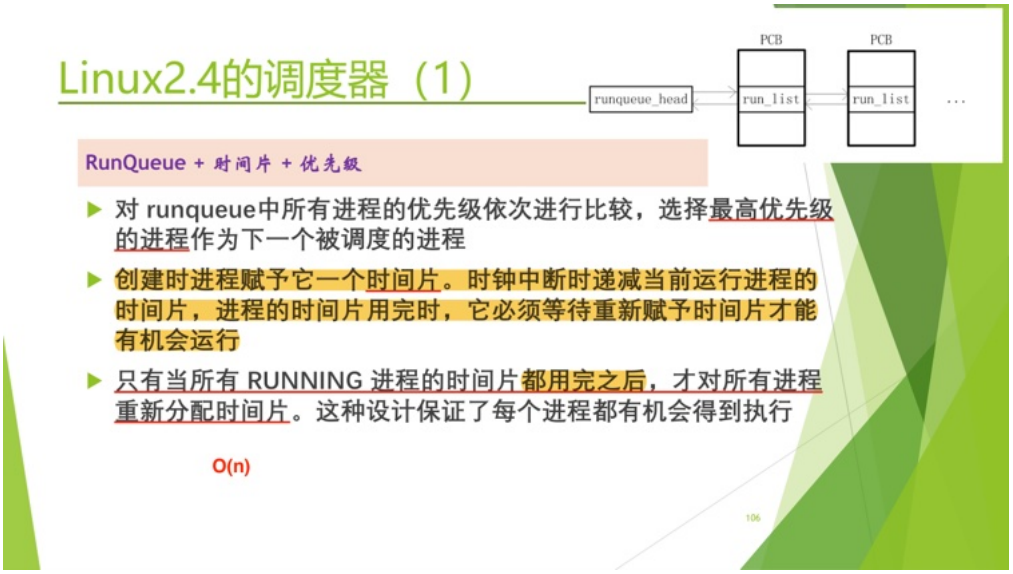
4.15.4 O(1) 调度器

- 普通进程动态优先级：`max(100, min(static_prio - bonus + 5, 139))`；
- 实时进程优先级不会动态修改；总比普通进程高；
- 平均睡眠时间用来判断是否是交互式进程。

- active 数组：当前可调度的进程，按优先级组织；
- expired 数组：用完时间片后被移过来的进程；
- pick next 时直接从 active 中按位图找最高优先级队列；
- active 空时交换 active 与 expired 指针 -> O(1) 复杂度；



图示：实时进程用 FIFO / RR，普通进程用 CFS。



图示：runqueue 上扫描所有进程比较优先级，时间片用完才统一重新分配。

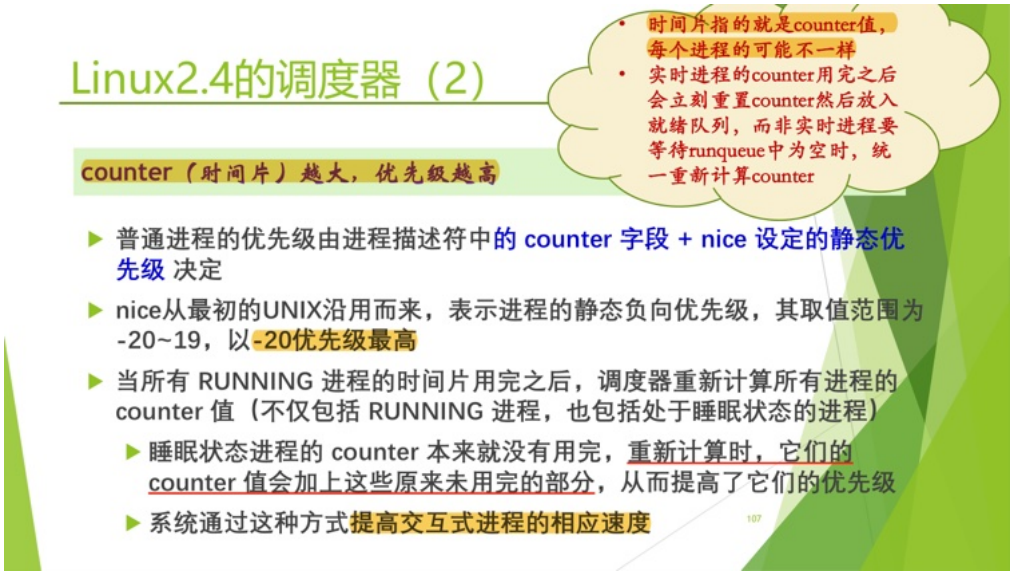
- 占用 CPU 时间过久的实时 / 交互进程也会被移到 expired 数组防止饿死其他进程。

4.15.5 SD（楼梯调度）

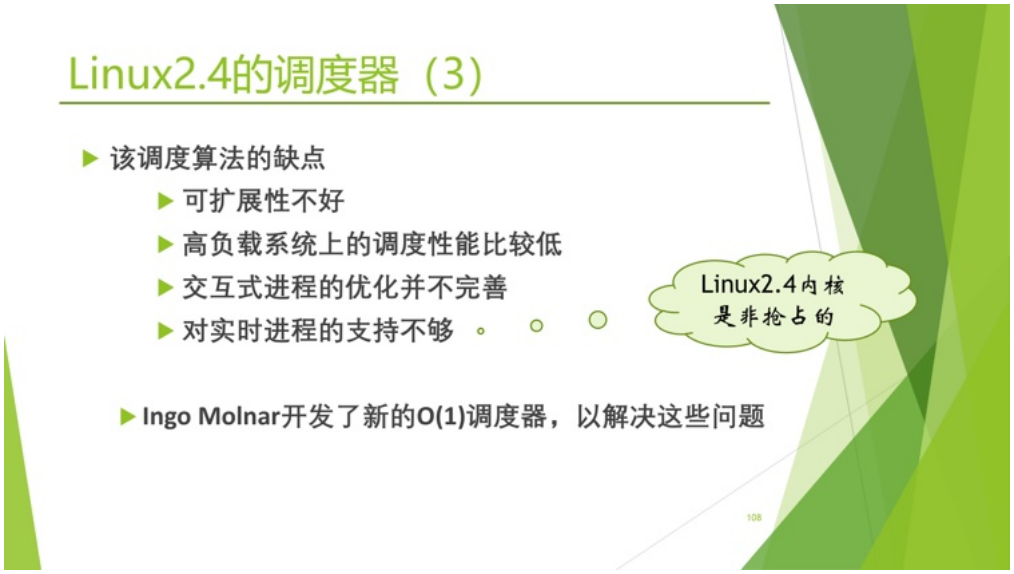
- 核心思想：完全公平，靠“楼梯”数据结构而非动态优先级；
- 用完时间片下楼梯一级；
- 到底后回到初始优先级下一级，并分配 N+1 倍时间片；
- 醒来的进程相对位置高，能更快被调度；
- 实时进程仍用 FIFO / RR。

4.15.6 RSDL

- 引入“**完全公平 + 可预测延迟**”：低优先级任务等待时间有上限。



图示：counter（剩余时间片）+ nice（静态优先级）共同决定优先级；睡眠进程的剩余 counter 增加。



图示：可扩展性差、高负载性能低、交互优化不完善、对实时支持不足。

4.15.7 CFS：完全公平调度器

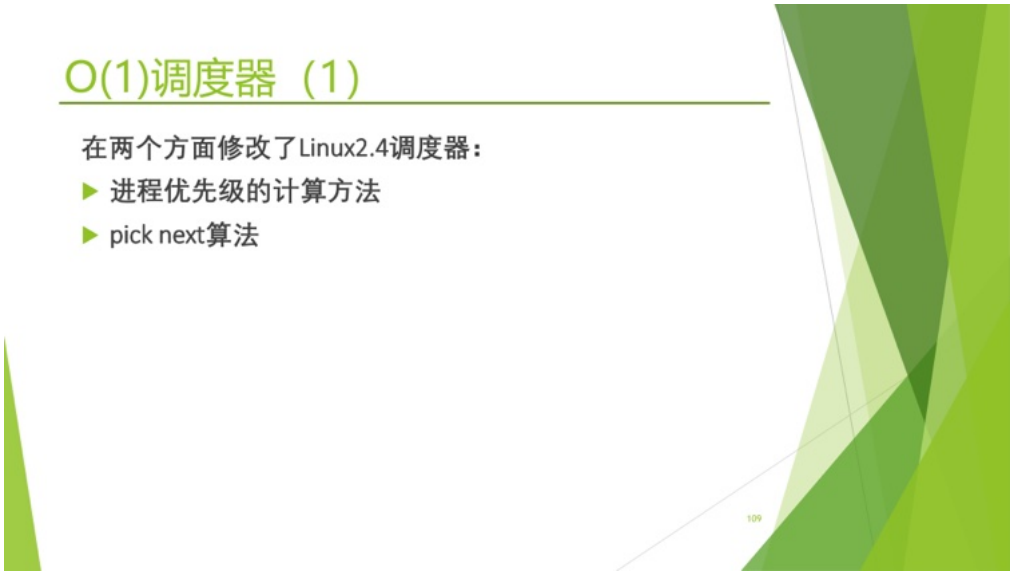
CFS 算法概述

公式 1：

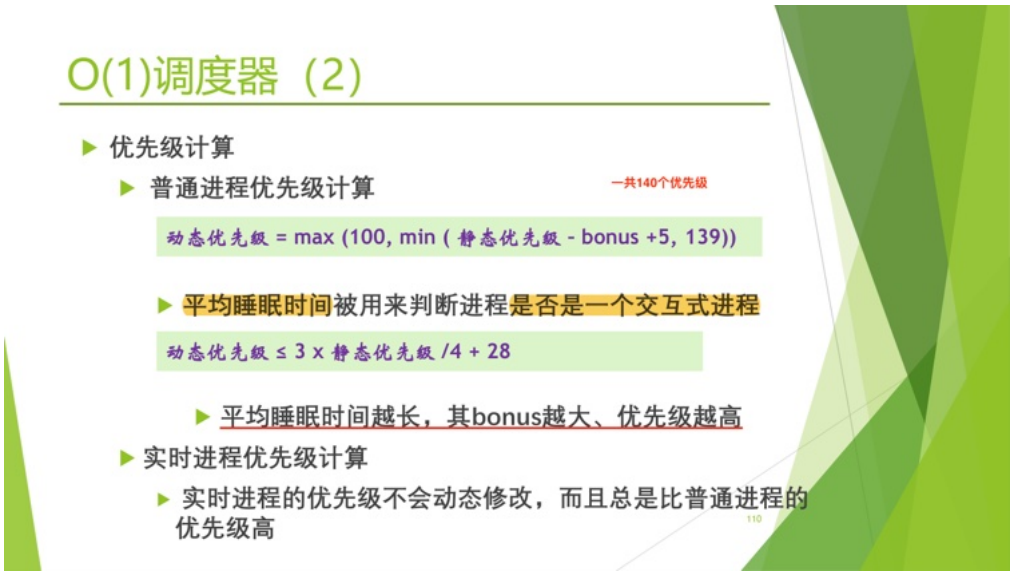
$$\text{分配运行时间} = \text{调度周期} \times \frac{\text{进程权重}}{\text{所有进程权重之和}}$$

公式 2：

$$\text{vruntime} = \text{实际运行时间} \times \frac{\text{NICE\_0\_LOAD}}{\text{进程权重}}$$



图示：在两方面修改 2.4：优先级计算与 pick next 算法。

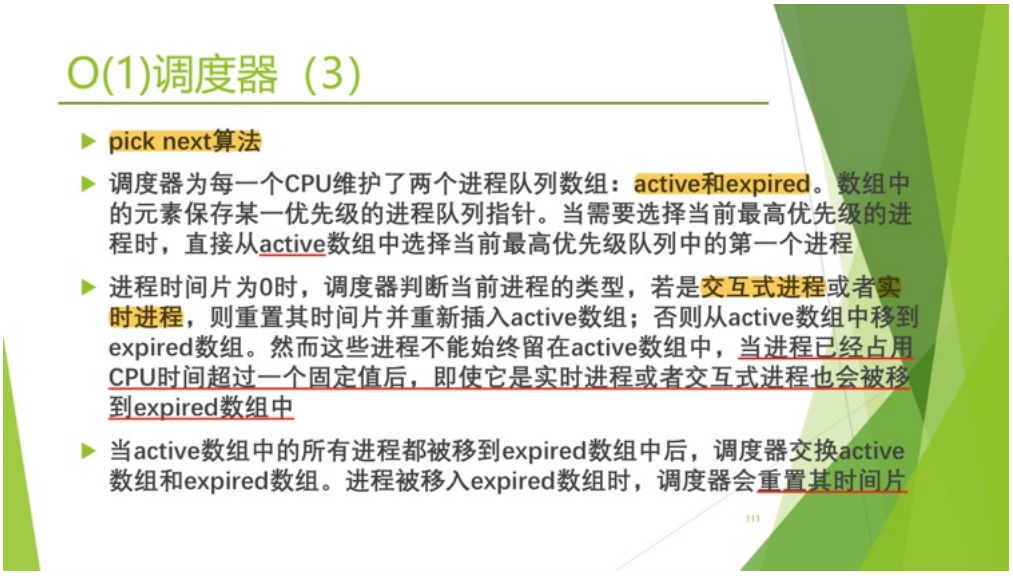


图示：动态优先级 $=\max(100, \min(\text{静态优先级} - \text{bonus} + 5, 139))$；总共 140 个优先级。

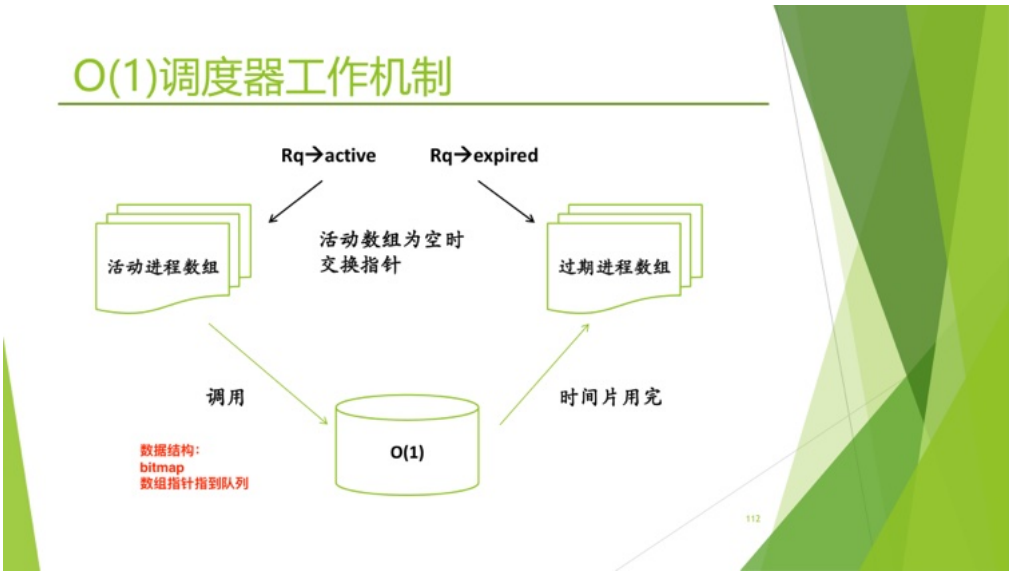
带入公式 1：所有进程在 vruntime 时间维度上分配的时间相等。CFS 每次选 vruntime 最小的进程运行。

数据结构

- `task_struct`：任务结构；
- `sched_entity`：调度实体，挂在红黑树上；
- `sched_class`：调度类（模块链），抽象不同调度算法的接口；
- `cfs_rq`：每个就绪队列保存红黑树根。



图示：每个 CPU 维护 active 与 expired 两个进程数组。



图示：active 指向当前可用，expired 指向过期；active 空时交换两个指针。

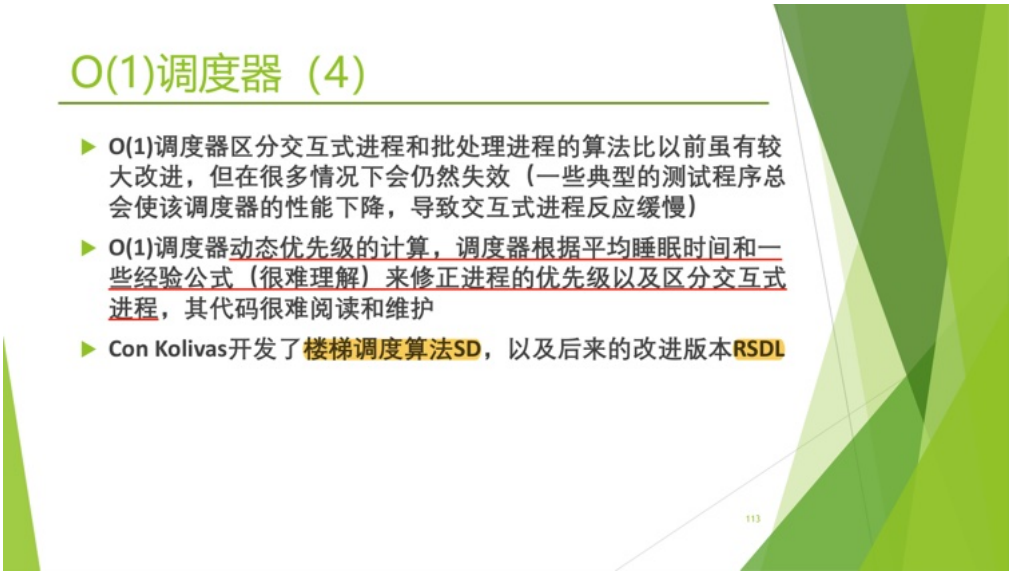
关键问题与解答

> **问 1：**新进程 vruntime 初值是不是 0?

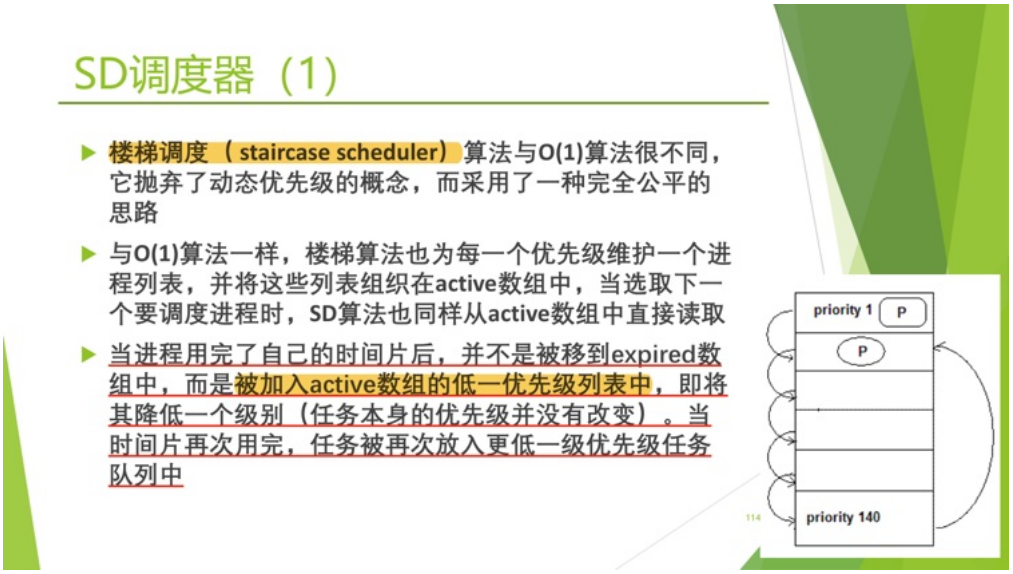
解答：不是。CFS 把新进程的 vruntime 初始化为当前就绪队列的 `min_vruntime`（可能再加一个小偏移）；如果设为 0，新进程会立刻抢占 CPU，破坏公平性。

> **问 2：**休眠进程 vruntime 一直保持不变吗?

解答：休眠期间保持不变，唤醒时可能被调整。如果休眠时间很长，醒来时 vruntime 仍是旧值会形成“补偿过度”，因此唤醒时取“计算出的补偿后 vruntime”和“原 vruntime”中的较大者。



图示：交互式判断仍易失效，动态优先级公式难以维护，催生了 SD/RSDL/CFS。



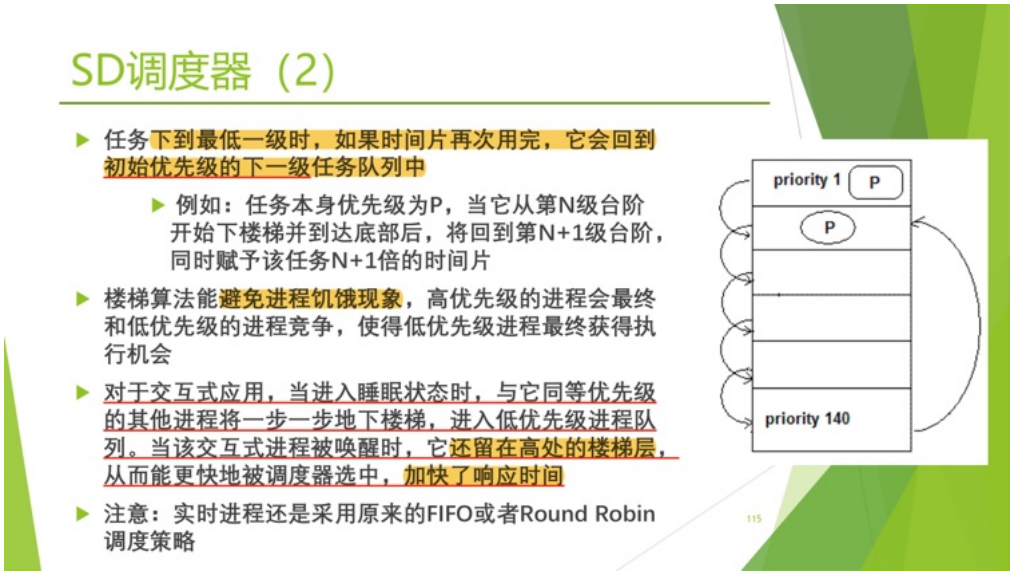
图示：抛弃动态优先级，引入“楼梯”思想，每用完时间片下降一级。

> **问 3：** 休眠进程在唤醒时立刻抢占 CPU 吗?

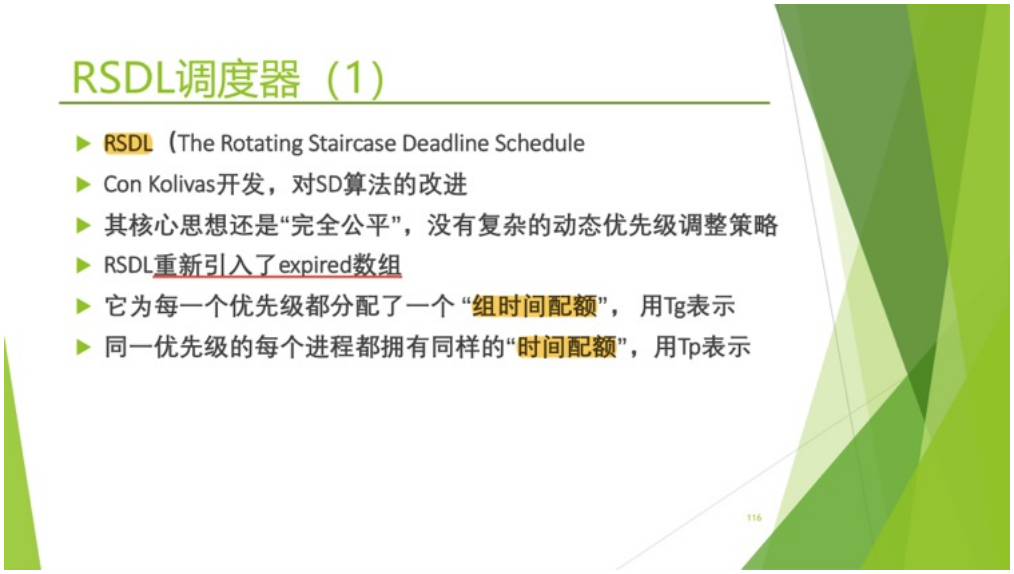
解答： 不一定。唤醒时进程比当前进程 vruntime 小则进入红黑树左侧；只有当 vruntime 差超过调度粒度（且满足其他条件）才会触发抢占。

> **问 4：** 进程迁移到另一个 CPU 时 vruntime 会变吗?

解答： 会调整。不同 CPU 上 `min_vruntime` 不同；迁移时通常用旧 CPU 的 vruntime 减去旧队列 `min_vruntime`，再加上新 CPU 的 `min_vruntime`，保持相对位置。



图示：到底后回到初始优先级的下一级，并赋予 N+1 倍时间片；睡眠期间其他线程下楼梯，使醒来的进程相对位置变高。



图示：在 SD 基础上引入 expired 数组、组时间配额 Tg、进程时间配额 Tp。

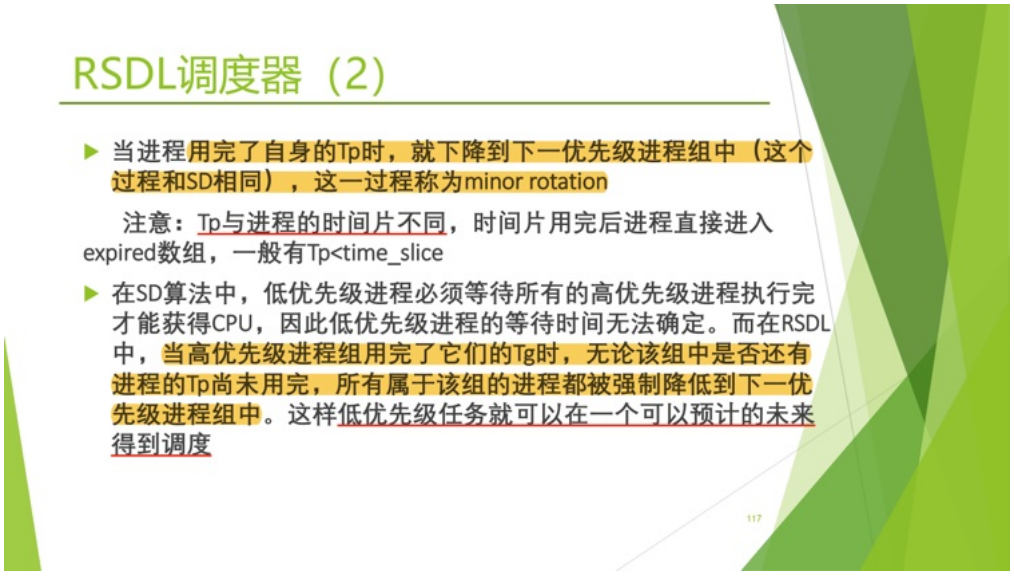
具体情景分析

时钟中断

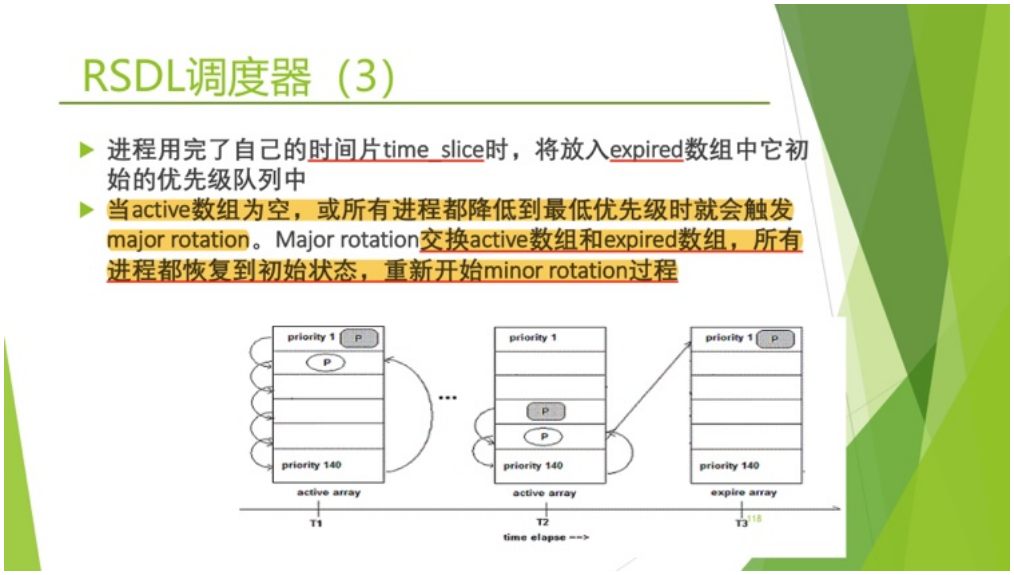
- 把当前进程这一段实际运行时间换算成 vruntime，加到 vruntime 上；
- 若当前进程已运行时间超过本调度周期分配给它的时间，做抢占标记，随后调用调度。

主动调度

- `cfs_rq->next` 指向被调度上 CPU 的进程；
- `cfs_rq->last` 指向最近被调度出去的进程；



图示：minor rotation：进程用完 Tp 下降一级；当组时间配额用完时所有进程都强制下降。

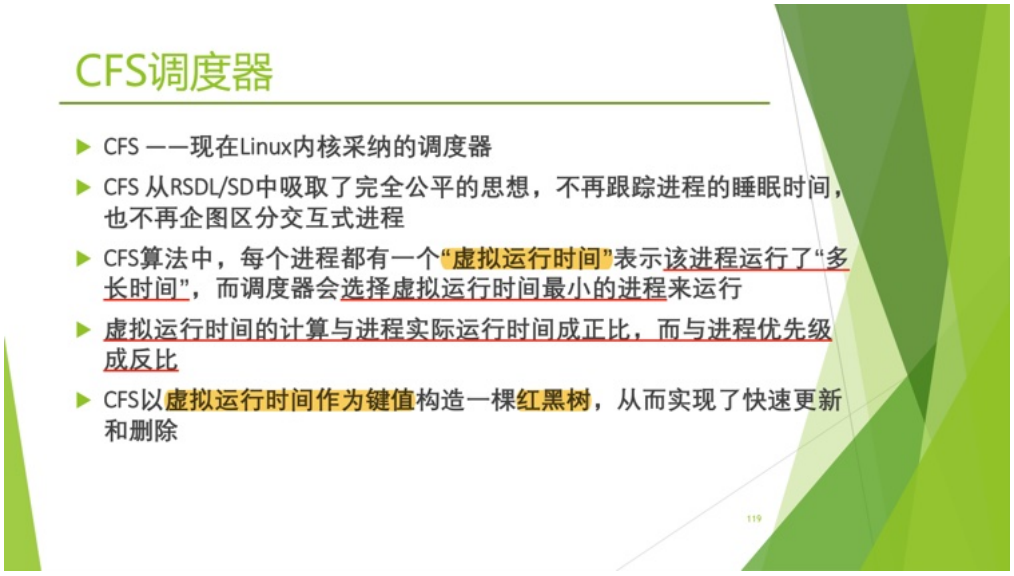


图示：进程用完时间片放入 expired；active 空或都到最低优先级时触发 major rotation 交换 active / expired。

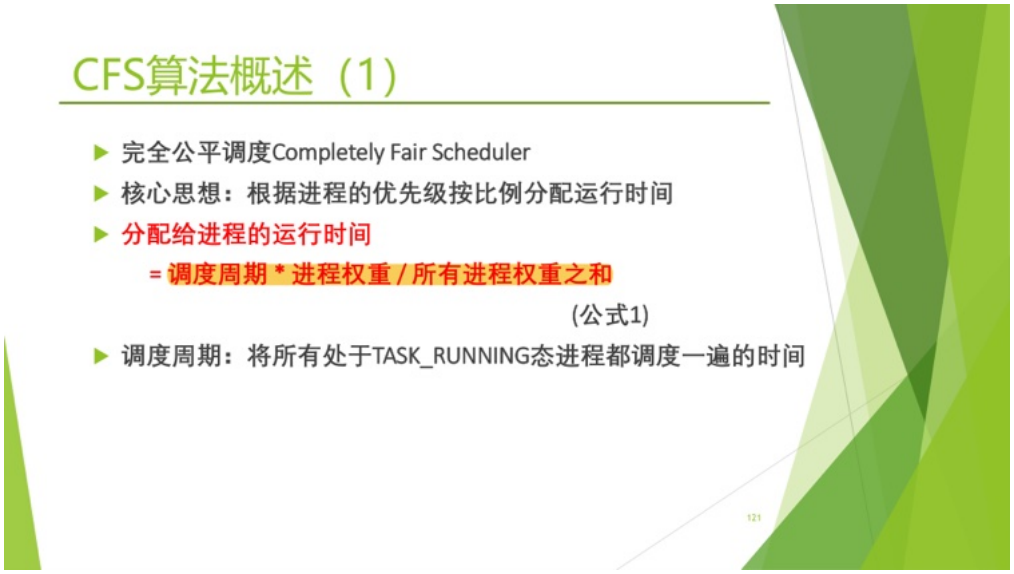
- 选下一个进程时优先选 next，再选 last，最后退化到红黑树最左结点；
- 通过 `wakeup_preempt_entity` 判断是否满足调度粒度（−1 / 0 / 1 三种结果）。

进程创建

- 新进程 vruntime = `min_vruntime` + 一次调度周期内能增加的 vruntime；
- 当 `sysctl` 设置了子进程优先运行、父子进程在同一 CPU、父进程 vruntime 小于子进程 vruntime 时，交换它们的 vruntime；
- 把新建进程插入红黑树；
- 根据 `wakeup_preempt_entity` 判断是否抢占当前进程。



图示：CFS 不再跟踪睡眠时间，不再判断交互式；用 vruntime 排序，红黑树维护。



图示：分配给进程的运行时间 = 调度周期 × 进程权重 / 所有进程权重之和。

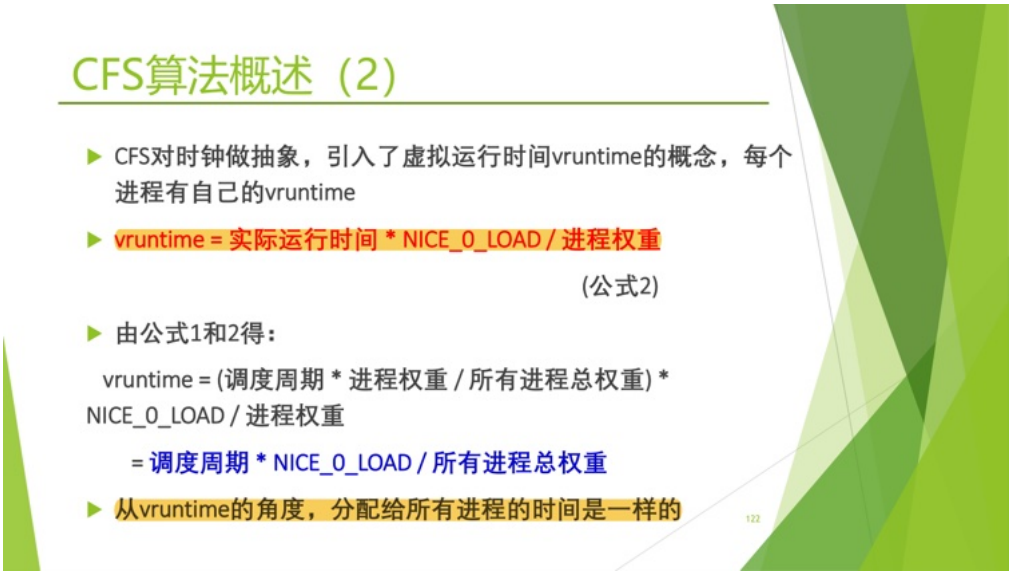
进程唤醒

- 用 `min_vruntime - sleeper_credit` 更新唤醒进程的 vruntime；
- 取“计算出的补偿后 vruntime”与“原 vruntime”的较大者，避免短睡眠获得不合理优惠；
- 插入红黑树并判断是否抢占。

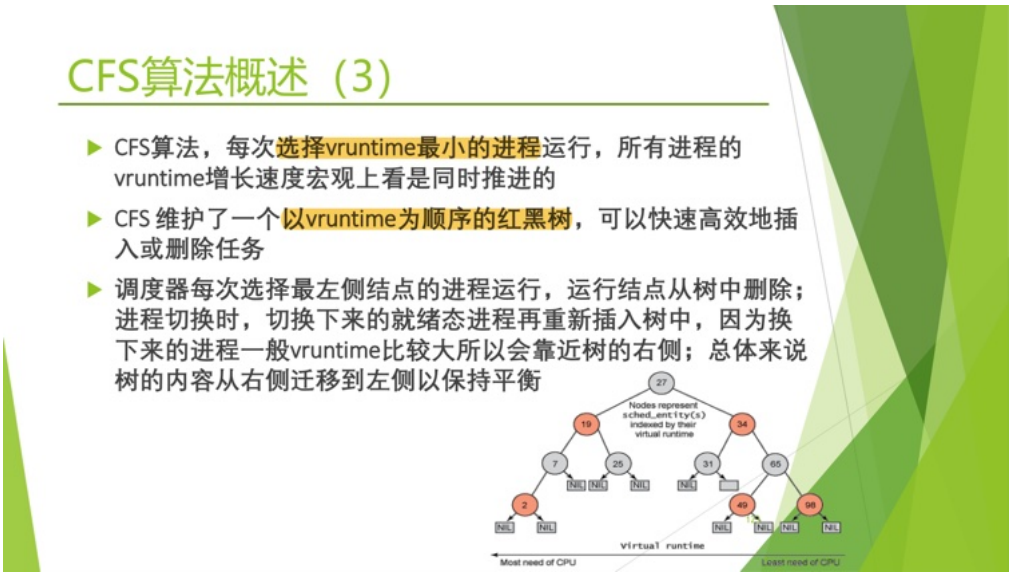
与进程状态转换的对应

记忆要点：

- 新建进程：计算 vruntime -> 插入红黑树；
- 唤醒进程：更新 vruntime -> 插入红黑树；



图示：vruntime = 实际运行时间 × NICE_0_LOAD / 进程权重。



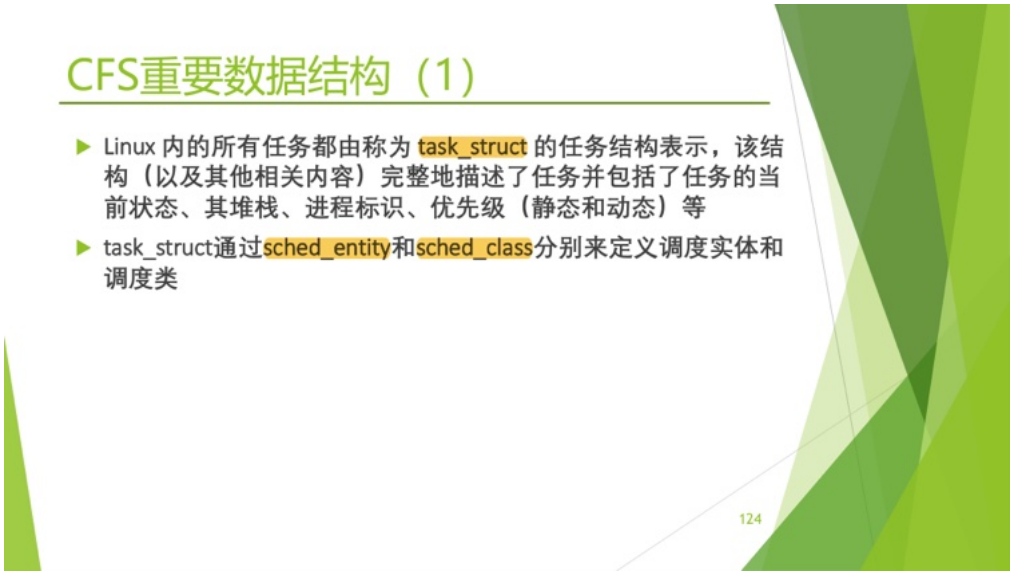
图示：以 vruntime 为键的红黑树；调度时选最左端结点；切换下来的进程重新插入红黑树。

- 时钟中断：更新时间片 / vruntime -> 检测是否抢占；
- schedule：选红黑树最左下结点。

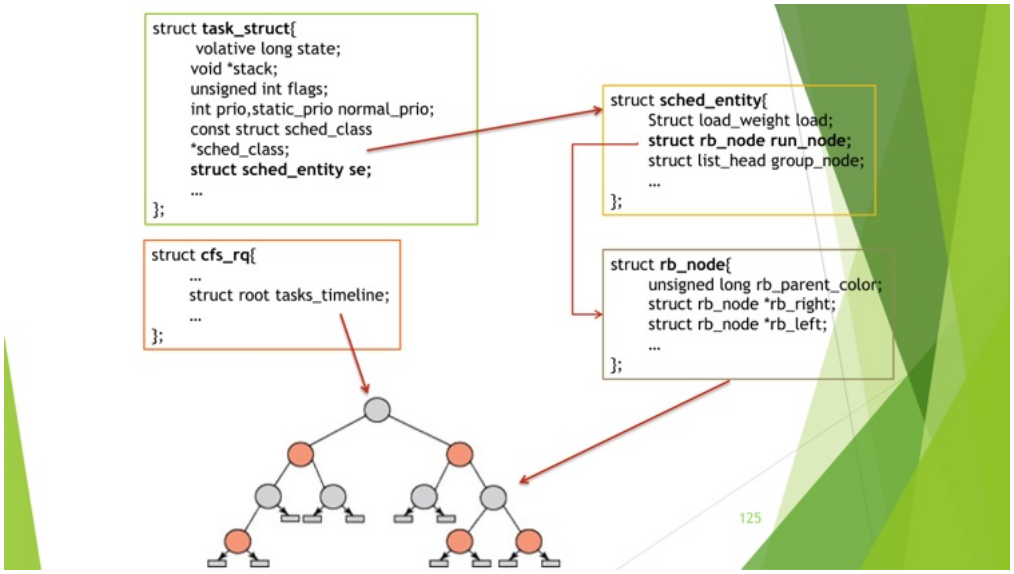
4.16 重点小结

记住几条主线：

- 调度三层次（长程 / 中程 / 短程）解决不同时间尺度的问题。
- 短程调度的三件事：What / When / How。
- 衡量调度算法的关键指标是吞吐量、周转时间、响应时间、CPU 利用率、公平性、可预测性。
- 经典算法：FCFS、SJF / SRTN、HRRN、RR / 虚拟 RR、优先级、多级反馈队列、公平共享、彩票、CFS、EDF / RM。



图示：task_struct + sched_entity + sched_class + cfs_rq + rb_node 之间的关系。



图示：cfs_rq、task_struct、sched_entity、rb_node 字段截取。

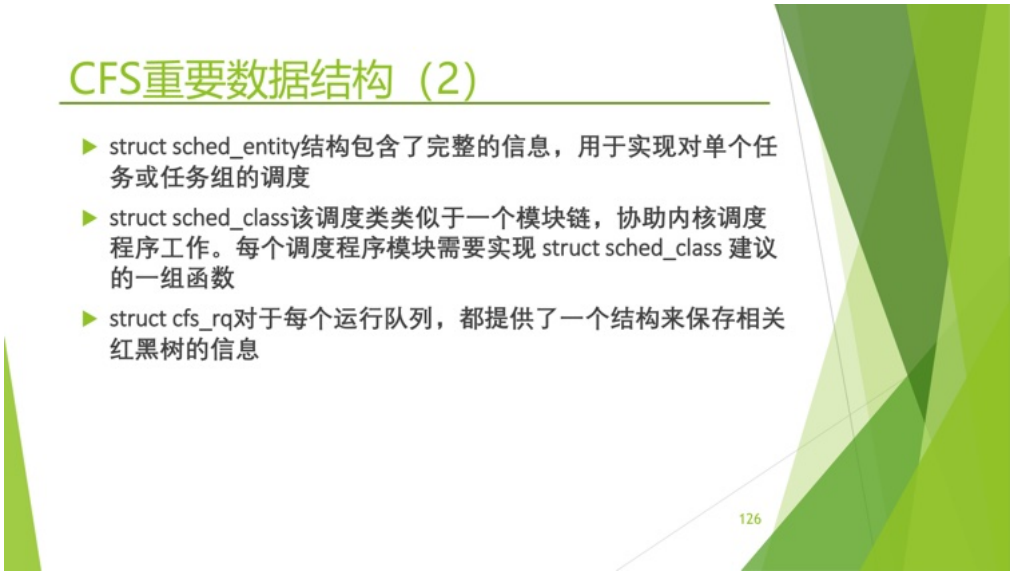
- 真实系统的调度器通常组合多种思想：Windows 抢占式优先级 + 时间配额；Linux CFS（普通）+ FIFO/RR（实时）。
- 多核调度引入新维度：缓存亲和性、负载均衡、迁移开销、Work Stealing。
- “机制 / 策略分离”贯穿全章：上下文切换、PCB、就绪队列是机制，算法是策略。

4.17 XV6 源代码阅读

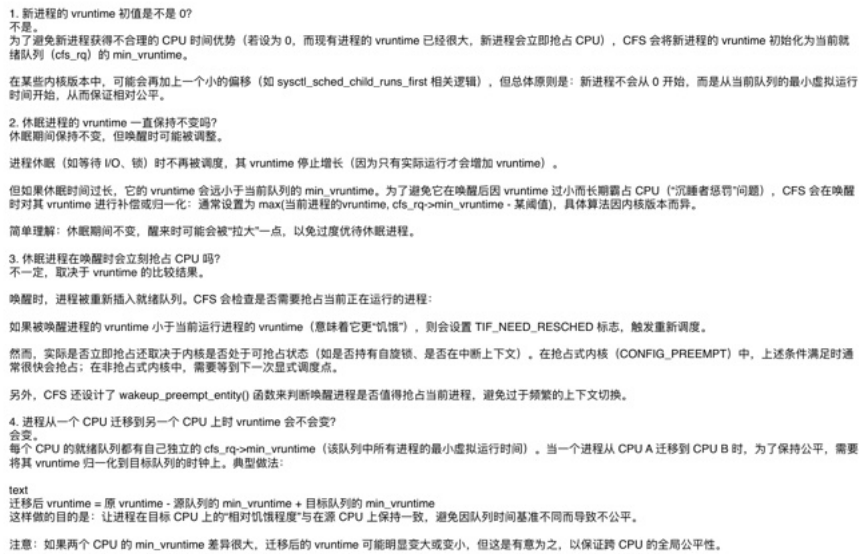
> **代码阅读题：**阅读 XV6 中 `swtch.S`、`proc.h`、`proc.c`，说明 XV6 的调度框架与上下文切换。

解答：

1. `struct proc` 与 `struct context`（`proc.h`）：



图示：新进程 vruntime 初值、休眠进程 vruntime 是否变、唤醒是否立即抢占、CPU 迁移是否变化。


1. 新进程的 vruntime 初值是不是 0?
不是。
为了避免新进程获得不合理的 CPU 时间优势（若设为 0，而现有进程的 vruntime 已经很大，新进程会立即抢占 CPU），CFS 会将新进程的 vruntime 初始化为当前就绪队列（cfs_rq）的 min_vruntime。

在某些内核版本中，可能会再加上一个小的偏移（如 sysctl_sched_child_runs_first 相关逻辑），但总体原则是：新进程不会从 0 开始，而是从当前队列的最小虚拟运行时间开始，从而保证相对公平。

2. 休眠进程的 vruntime 一直保持不变吗?
休眠期间保持不变，但唤醒时可能被调整。

进程休眠（如等待 I/O、锁）时不再被调度，其 vruntime 停止增长（因为只有实际运行才会增加 vruntime）。

但如果休眠时间过长，它的 vruntime 会远小于当前队列的 min_vruntime。为了避免它在唤醒后因 vruntime 过小而长期霸占 CPU（"沉睡者惩罚"问题），CFS 会在唤醒时对其 vruntime 进行补偿或归一化：通常设置为 max(当前进程的vruntime, cfs_rq->min_vruntime - 某阈值)，具体算法因内核版本而异。

简单理解：休眠期间不变，醒来时可能会被"拉大"一点，以免过度优待休眠进程。

3. 休眠进程在唤醒时会立刻抢占 CPU 吗?
不一定，取决于 vruntime 的比较结果。

唤醒时，进程被重新插入就绪队列。CFS 会检查是否需要抢占当前正在运行的进程：

如果被唤醒进程的 vruntime 小于当前运行进程的 vruntime（意味着它更"饥饿"），则会设置 TIF_NEED_RESCHED 标志，触发重新调度。

然而，实际是否立即抢占还取决于内核是否处于可抢占状态（如是否持有自旋锁、是否在中断上下文）。在抢占式内核（CONFIG_PREEMPT）中，上述条件满足时通常很快会抢占；在非抢占式内核中，需要等到下一次显式调度点。

另外，CFS 还设计了 wakeup_preempt_entity() 函数来判断唤醒进程是否值得抢占当前进程，避免过于频繁的上下文切换。

4. 进程从一个 CPU 迁移到另一个 CPU 上时 vruntime 会不会变?
会变。
每个 CPU 的就绪队列都有自己独立的 cfs_rq->min_vruntime（该队列中所有进程的最小虚拟运行时间）。当一个进程从 CPU A 迁移到 CPU B 时，为了保持公平，需要将其 vruntime 归一化到目标队列的时钟上。典型做法：

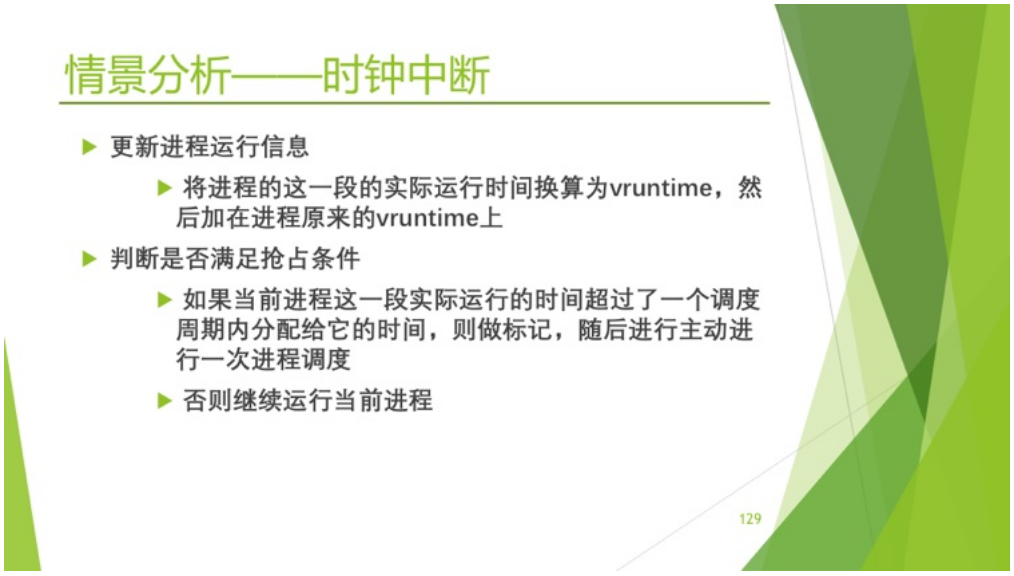
text
迁移后 vruntime = 原 vruntime - 源队列的 min_vruntime + 目标队列的 min_vruntime
这样做的目的是：让进程在目标 CPU 上的"相对饥饿程度"与在源 CPU 上保持一致，避免因队列时间基准不同而导致不公平。

注意：如果两个 CPU 的 min_vruntime 差异很大，迁移后的 vruntime 可能明显变大或变小，但这是有意为之，以保证跨 CPU 的全局公平性。

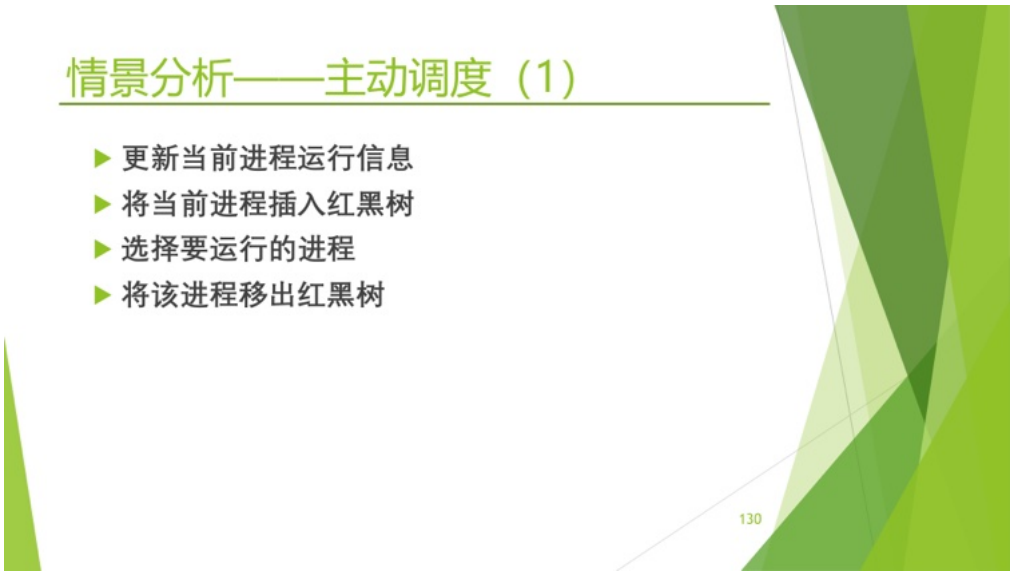
图示：时钟中断、主动调度、进程唤醒、进程创建。

- `struct context`：保存内核切换需要的寄存器（`ra`，`sp`，`s0~s11`），是 `swtch` 的输入输出。
- `struct proc`:进程的所有状态——`state`，`pid`，`parent`，`pagetable`，`sz`，`kstack`，`trapframe`，`context`，`name` 等。

2. **`swtch(old, new)`（`swtch.S`）**：
   - 入参 `a0` 指向 `old context`，`a1` 指向 `new context`；
   - 把 `ra`，`sp`，`s0~s11` 保存到 `old`，再从 `new` 读回；
   - `ret` 跳到新进程上次保存的 `ra`，从而回到对应的执行点。
3. **调度循环（`scheduler in proc.c`）**：
   - 每个 hart 有自己的调度循环；



图示：更新当前进程 vruntime；判断是否满足抢占条件。

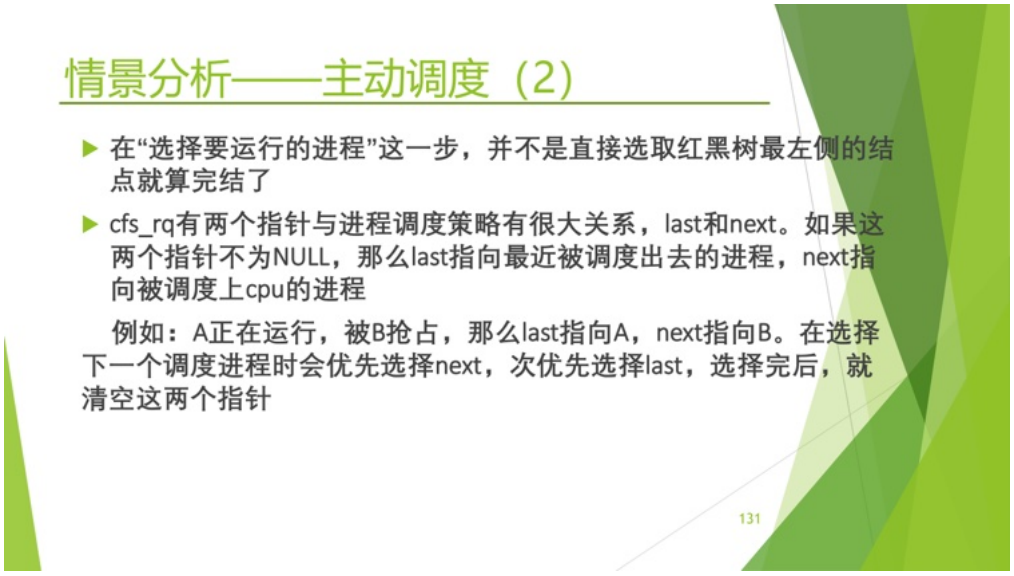


图示：更新 vruntime；将当前进程插入红黑树；选下一个进程；移出红黑树。

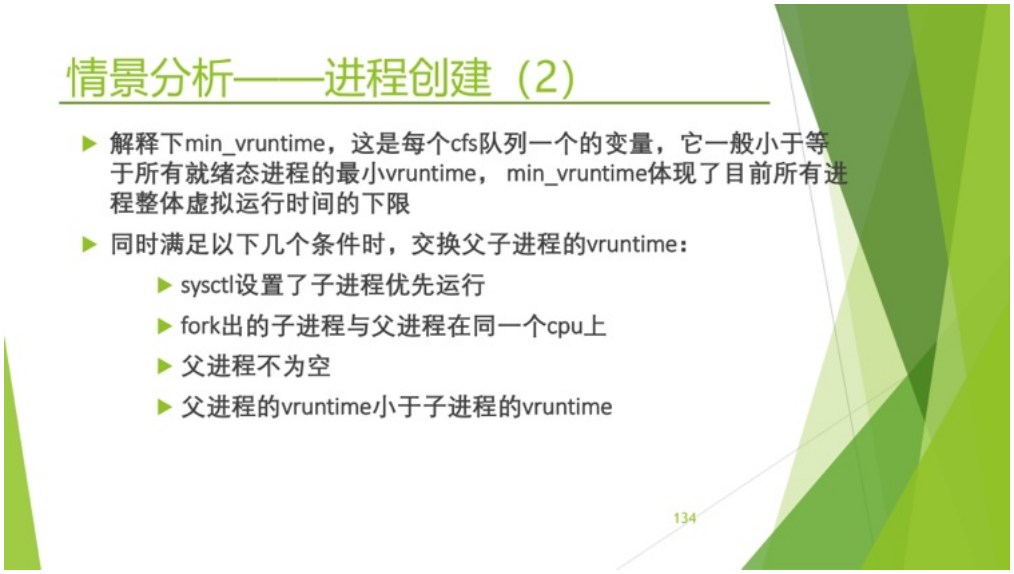
- 关中断，遍历进程表，找到 `RUNNABLE` 进程：
  - 设置当前 CPU 的 `c->proc = p`；
  - 状态置为 `RUNNING`；
  - 调用 `swtch(&c->context, &p->context)` 切到该进程的内核上下文；
- 进程让出 CPU 时通过 `sched()` 切回调度器上下文，回到调度循环继续找下一个就绪进程。

4. 进程让出 CPU（`sched`、`yield`）：

- `yield()`：在时钟中断中调用，把当前进程状态置为 `RUNNABLE` 并 `sched`；
- `sched()`：检查锁、关中断状态、`p->state` 等不变量，然后 `swtch(&p->context, &c->context)` 切回调度器；
- `sched` 返回时表示进程被重新调度上 CPU。

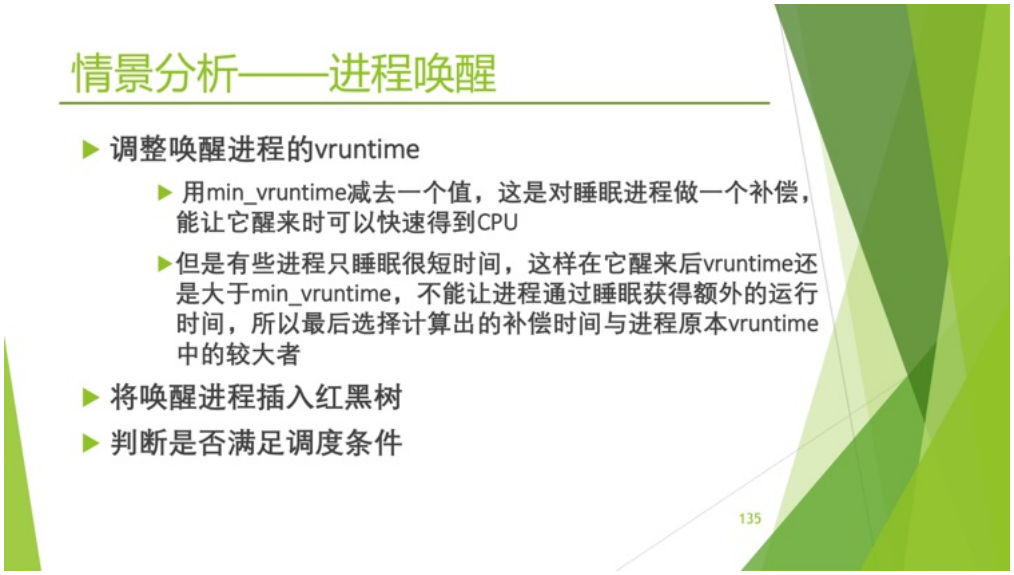


图示：cfs_rq->next 与 cfs_rq->last 的优先选择，避免立刻反复切换。

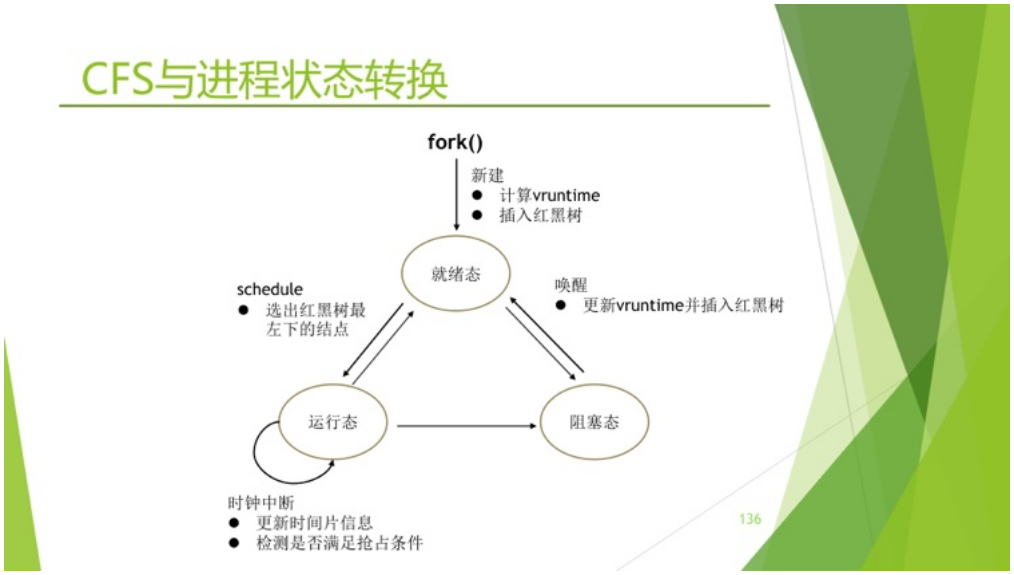


图示：父子进程 vruntime 交换条件、`min_vruntime` 含义。

5. **进程切换的整体路径**：
   - 用户态 -> 时钟中断进入内核 -> `usertrap` 调用 `yield` -> `sched` -> `swtch` 切到调度器；
   - 调度器找到下一个 `RUNNABLE` 进程，`swtch` 切到它的内核上下文；
   - 该进程在 `sched` 中“返回”，继续处理剩余路径，最终 `usertrapret` 切回用户态。
6. **简化与真实差异**：
   - XV6 没有 vruntime / 红黑树，调度策略是简单的“轮询表 + RUNNABLE”；
   - 没有抢占式优先级、亲和性、负载均衡；
   - 但完整体现了“**进程切换 = 上下文切换 + 状态机维护**”的核心思想，是理解 Linux/Windows 调度复杂实现的良好起点。



图示：唤醒时调整 vruntime，对长睡眠做有限补偿。



图示：从 fork、唤醒、时钟中断到 schedule，CFS 的关键操作和红黑树的更新。



图示：基本概念、调度层次、进程行为、设计目标、典型算法、Linux/Windows 调度、调度时机与切换过程。



图示：阅读范围为 `kernel/swtch.S`、`kernel/proc.h`、`kernel/proc.c`，对应 xv6-book 第 7.1–7.4 节。

LECTURE 5

第 5 讲

内存管理概述

Operating Systems Course Notes

5.1 本讲导读

- 这一讲解决两个核心问题：
  1. 内存如何在多个进程之间被合理分配、保护与共享?
  2. 如何把进程的逻辑地址转换为物理内存地址?
- 知识脉络：
  1. 历史演化与基本概念（地址重定位、保护、地址空间）；
  2. 物理内存管理（数据结构、分配算法、伙伴系统、SLAB/SLUB/SLOB）；
  3. 基本内存管理方案：单一连续区、固定分区、可变分区、分页、分段、段页式；
  4. 内存“扩充”技术：紧凑、覆盖、交换。
- 后续“虚拟存储技术”讲是在分页方案之上的进一步扩展，本讲是它的基础。

> **关键提醒：**内存管理 ≠ 虚拟内存管理。本讲讲“没虚拟内存之前怎么管”，下一讲讲“引入虚拟内存后怎么管”。



图示：内存管理 ≠ 虚拟内存管理；本讲讨论物理内存管理与基本内存方案。

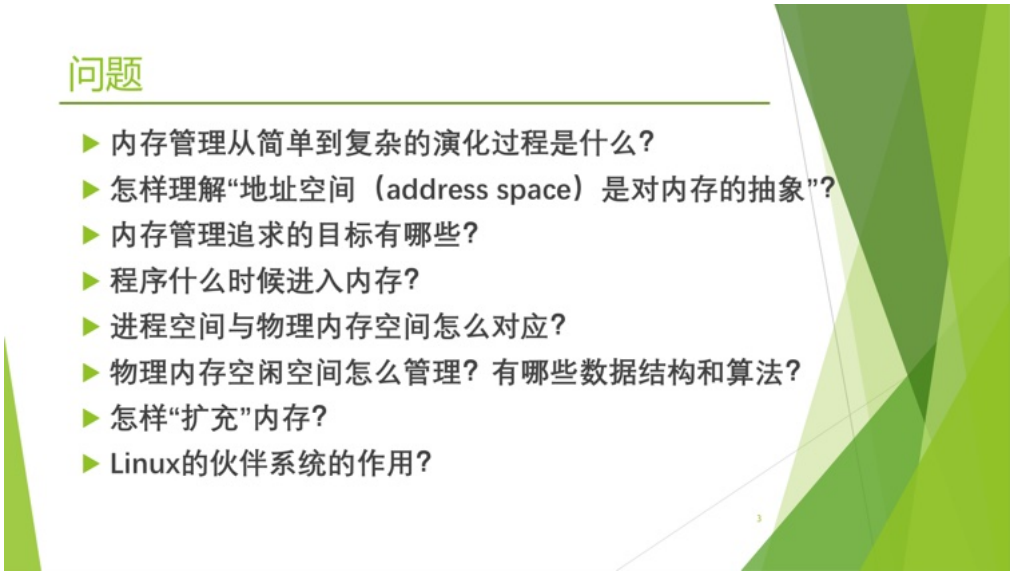
5.2 开篇问题与重要概念

> **问 1：**内存管理从简单到复杂的演化过程是什么?

解答：早期“一次只有一个程序使用全部内存” -> 多道程序设计（多个程序同时驻留）-> 分时共享 -> 引入虚拟内存。每一步演化都来源于资源利用率提升和共享需求增长。

> **问 2：**怎样理解“地址空间是对内存的抽象”?

解答： OS 给每个进程提供一段独立的虚拟地址范围，让程序使用统一的逻辑地址，不必关心物



图示：内存管理演化、地址空间、目标、装入时机、空闲管理、伙伴系统等问题。

理内存的位置和形态。底层通过地址转换 / 重定位完成。它隔离进程、便于扩展、便于共享与保护。

> **问 3：**内存管理追求哪些目标?

解答：抽象（让程序不依赖具体物理位置）、效率（减少访问开销）、保护（隔离进程、防越界 / 越权）、共享（合理共享只读代码、共享库等）、最小化访问时间。

> **问 4：**程序什么时候进入内存?

解答：早期是装载时全部进入；现代多采用按需调入（demand paging）。绑定时机有四种：编译期、链接期、装载期、运行期。现代 OS 主流做法是**运行期绑定**（动态地址重定位），结合 MMU。

> **问 5：**进程地址空间与物理内存怎么对应?

解答：通过分页 / 分段 / 段页式机制做地址映射；硬件 MMU 完成转换；OS 维护页表 / 段表。

> **问 6：**物理内存空闲空间怎么管理?

解答：通过位图、空闲区表、空闲区链表等数据结构跟踪空闲区；用首次适配 / 最佳适配 / 最差适配 / 下次适配等算法分配；伙伴系统处理 2 的幂大小块；SLAB / SLUB / SLOB 处理小对象分配。详见 §5。

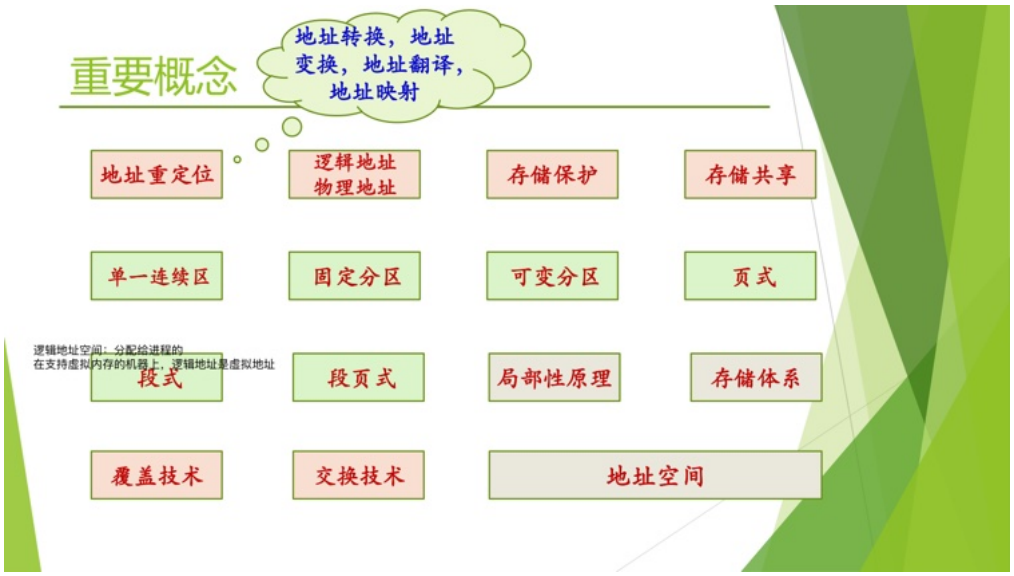
> **问 7：**怎样“扩充”内存?

解答：紧凑技术整理碎片；覆盖技术在程序内部分块切换；交换技术把不活跃进程换出磁盘；最终在虚拟存储中达到“按需调页 + 置换”的极致。

> **问 8：** Linux 伙伴系统的作用？

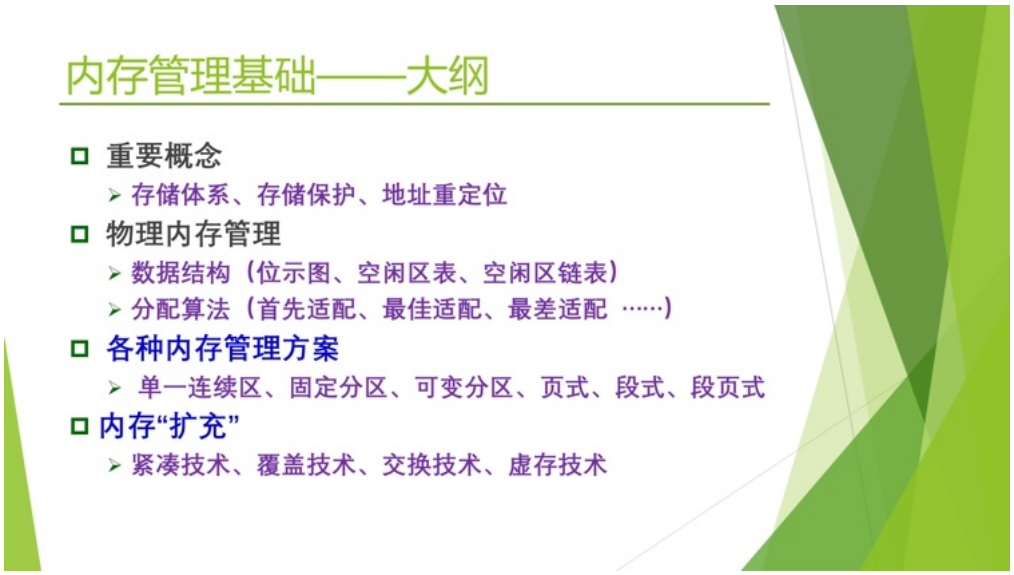
解答： Linux 内核通过伙伴系统管理物理页框，按 2 的幂分配；上层 SLAB/SLUB 再把伙伴系统分配的大块切成小对象，满足内核频繁分配小对象的需求。详见 §5.4 与 §5.5。

5.3 重要概念与本讲大纲



图示：地址重定位、逻辑地址、物理地址、保护、共享、单一连续区 / 固定分区 / 覆盖 / 可变分区 / 页式 / 段式 / 段页式 / 交换 / 局部性 / 存储体系。

> **要点：** 在支持虚拟内存的机器上，**逻辑地址（logical address）= 虚拟地址（virtual address）**；分配给进程的是“**逻辑地址空间**”。地址转换 / 地址变换 / 地址翻译 / 地址映射通常指同一件事。



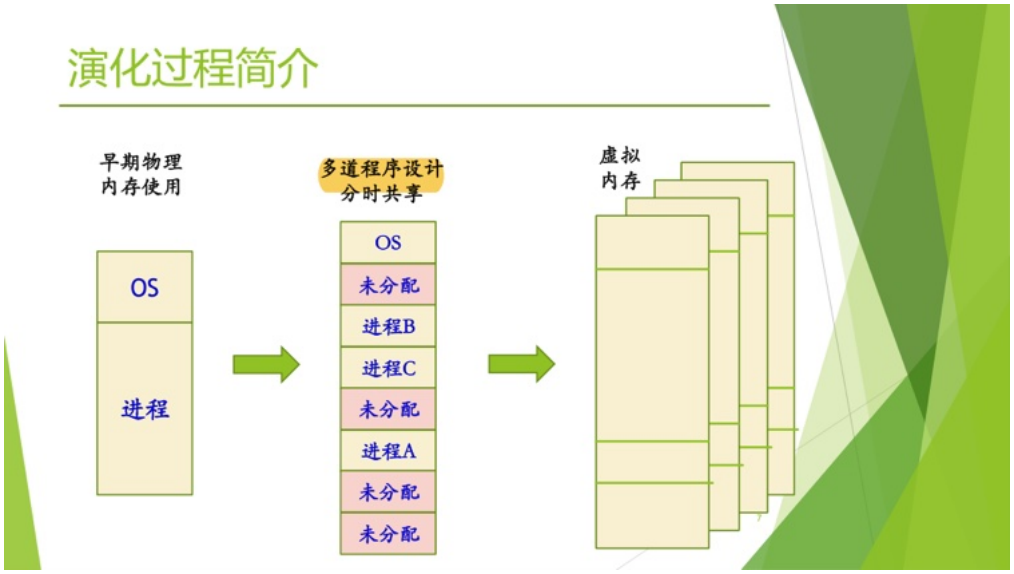
图示：重要概念 -> 物理内存管理 -> 各种内存管理方案 -> 内存“扩充”。

5.4 演化过程与基础概念



图示：地址重定位、地址保护是本讲的两个核心。

5.4.1 历史演化



图示：早期单程序占用 -> 多道程序设计 -> 分时共享 -> 虚拟内存。

- 早期物理内存：操作系统 + 单个进程；
- 多道程序设计：多个进程同时驻留，需要保护与隔离；
- 分时共享：多个用户、多个程序交替使用 CPU 与内存；
- 虚拟内存：把进程地址空间和物理内存解耦。

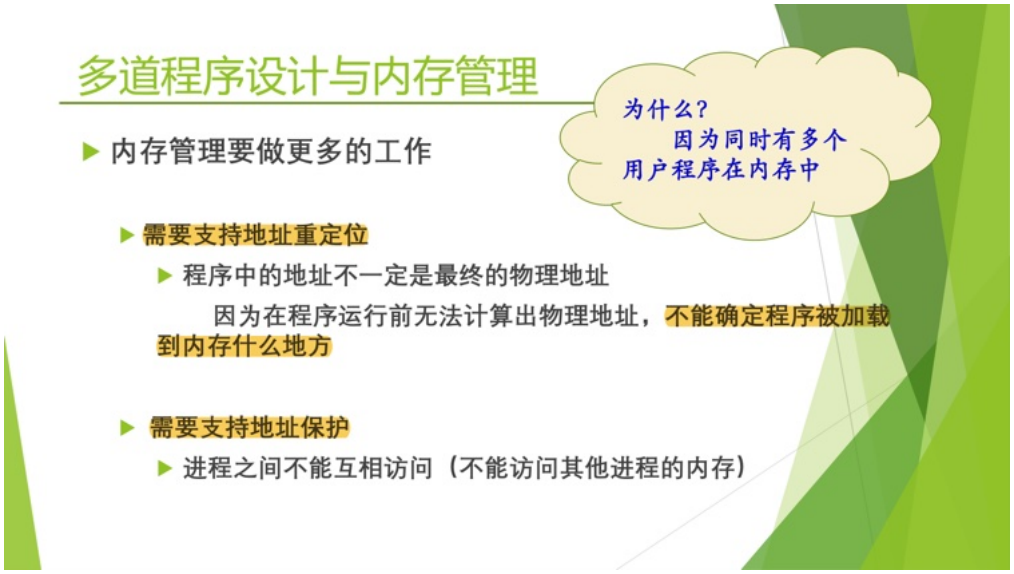
- 程序装载到内存才能运行；
- 程序以可执行文件格式保存在磁盘；



图示：可执行文件以文件形式存放；多道程序模型下进程有独立地址空间；存在存储体系；越界 / 越权保护。

- 多道程序模型让多个程序进入内存并行运行；
- 每个进程有独立地址空间，相互不能直接访问；
- **存储体系**：寄存器 -> L1/L2/L3 缓存 -> 内存 -> 本地磁盘 -> 远程磁盘；
- 越界保护、越权保护是基础需求。

5.4.2 多道程序设计带来的需求

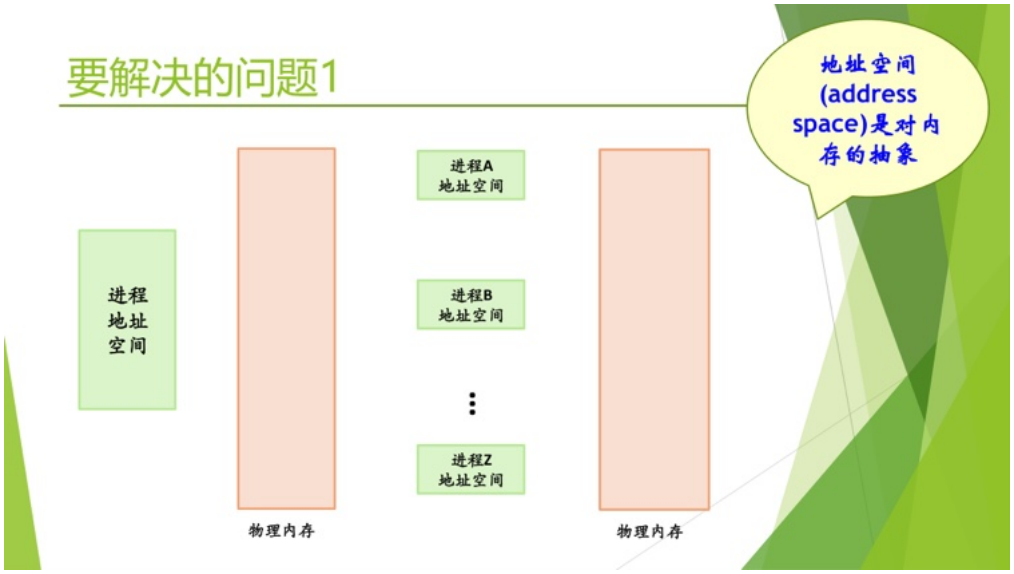


图示：内存管理需要支持地址重定位与地址保护，因为同时有多个用户程序在内存中。

- 多个程序同时存在 -> 必须支持地址重定位（程序的逻辑地址不再等于物理地址）；
- 必须支持地址保护，进程之间不能互相访问内存。



图示：每个进程拥有自己的地址空间，把物理内存抽象成多个独立的逻辑空间。



图示：进程 A、B、Z 各自的地址空间映射到同一物理内存的不同位置。

5.4.3 地址空间是对内存的抽象

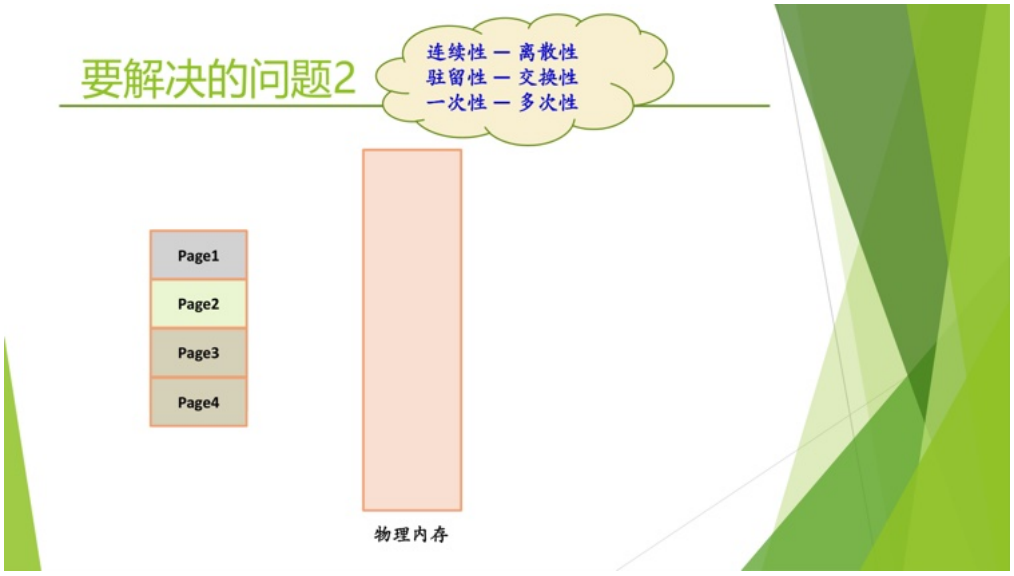
三对相对的概念：

- **连续性 vs 离散性**：进程的页是否在物理内存中连续；
- **驻留性 vs 交换性**：进程是否一直驻留内存；
- **一次性 vs 多次性**：进程一次性全部装入还是按需多次装入。

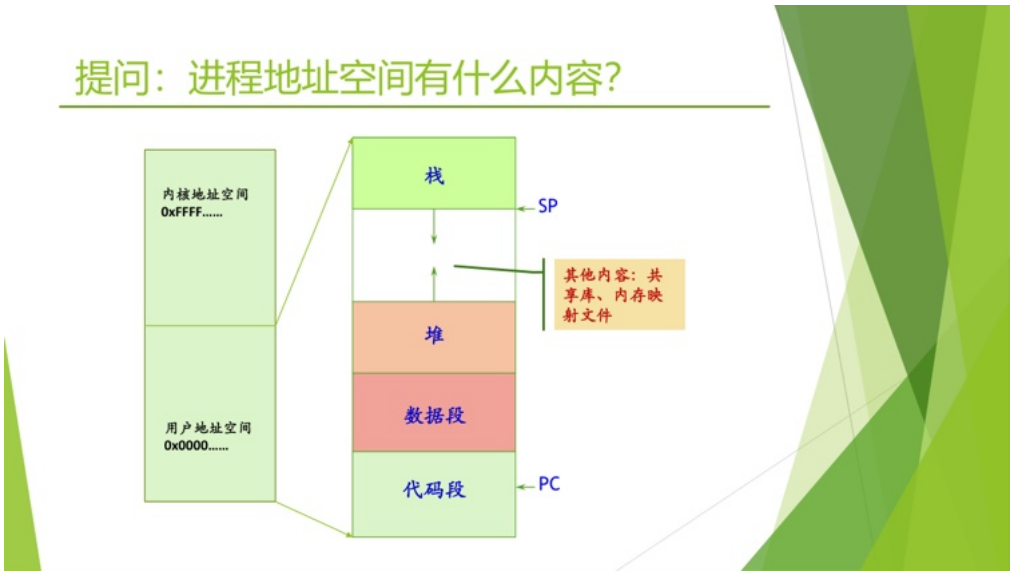
后续的页式、段页式、虚拟存储分别在这三个维度上选择更复杂的组合。

5.4.4 进程地址空间内容

补充几个相关名词：



图示：地址空间的连续性 vs 离散性、驻留性 vs 交换性、一次性 vs 多次性。

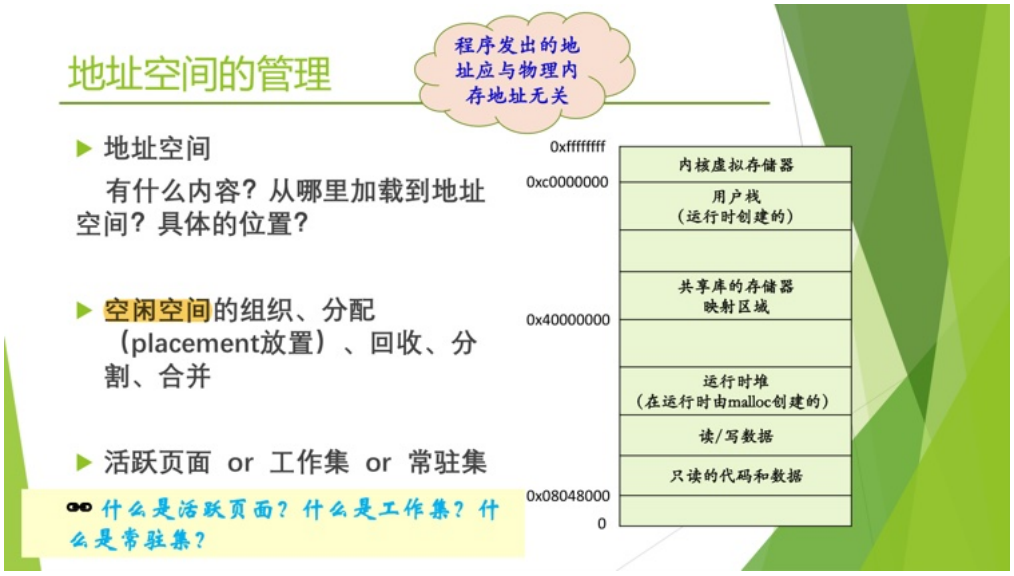


图示：典型布局——代码段、数据段、堆、栈、共享库、内存映射文件、内核地址空间。

- **活跃页面（Active Pages）**：当前正在被频繁访问的页面集合；
- **工作集（Working Set）**：在过去一段时间内访问过的页面集合，由 Denning 提出；
- **常驻集（Resident Set）**：当前驻留在物理内存的页面集合。它们不一定相同，但共同决定缺页率与替换策略。

5.4.5 XV6 的地址空间

- 布局简单，便于教学；
- trampoline 是“跨特权级跳板”，必须在用户态和内核态都映射相同位置，才能保证陷入 / 返回时 PC 平滑切换。



图示：地址空间的内容、加载来源、空闲管理、活跃页面 / 工作集 / 常驻集等问题。



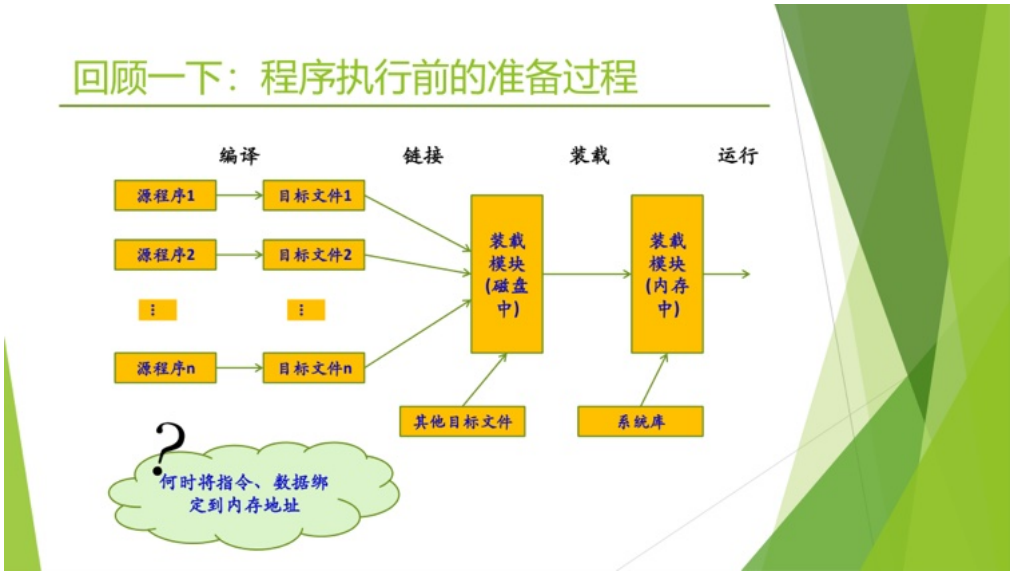
图示：text -> data/bss -> stack -> heap，高地址处放 trampoline / trapframe（用户态与内核态都映射）。

5.5 地址重定位与地址保护

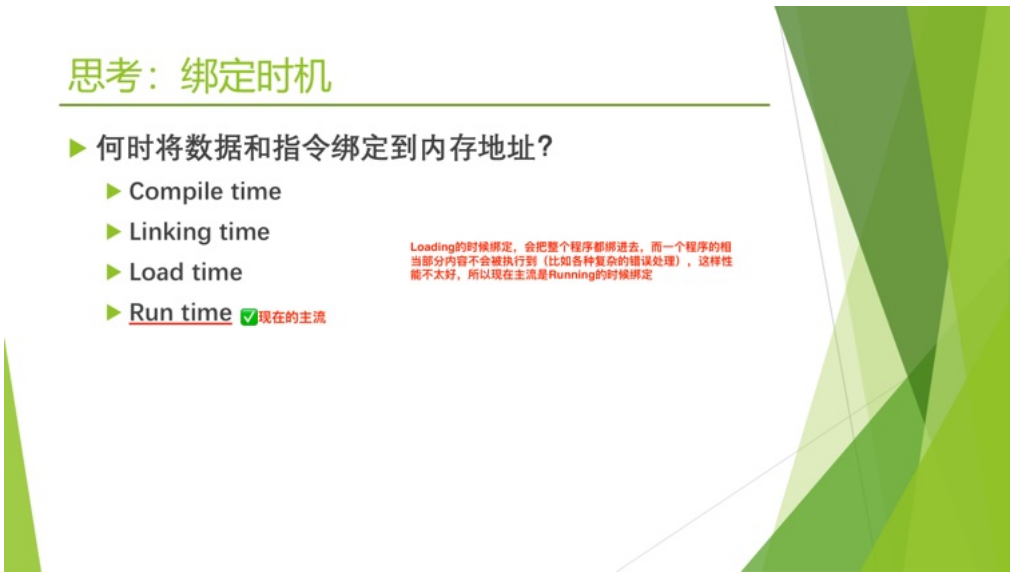
5.5.1 程序执行前的准备过程

> **思考：**何时把指令、数据绑定到内存地址?

- **编译期绑定**：编译时直接生成绝对地址（早期、嵌入式）；
- **链接期绑定**：链接时确定地址（部分嵌入式系统）；
- **装载期绑定**：把整个程序绑入物理地址（典型“静态重定位”）；
- **运行期绑定**：运行时按需绑定，使用 MMU 做动态地址重定位（**现代主流**）。



图示：源程序 -> 编译 -> 目标文件 -> 链接 -> 装载模块（磁盘）-> 装载 -> 装载模块（内存）-> 运行。



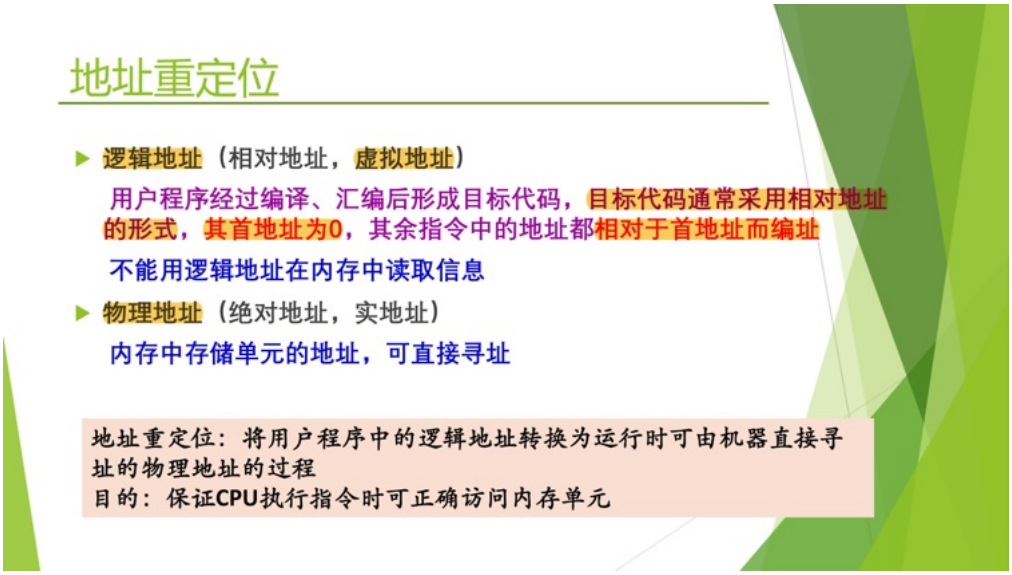
图示：编译期 / 链接期 / 装载期 / 运行期四种时机。

5.5.2 逻辑地址 vs 物理地址

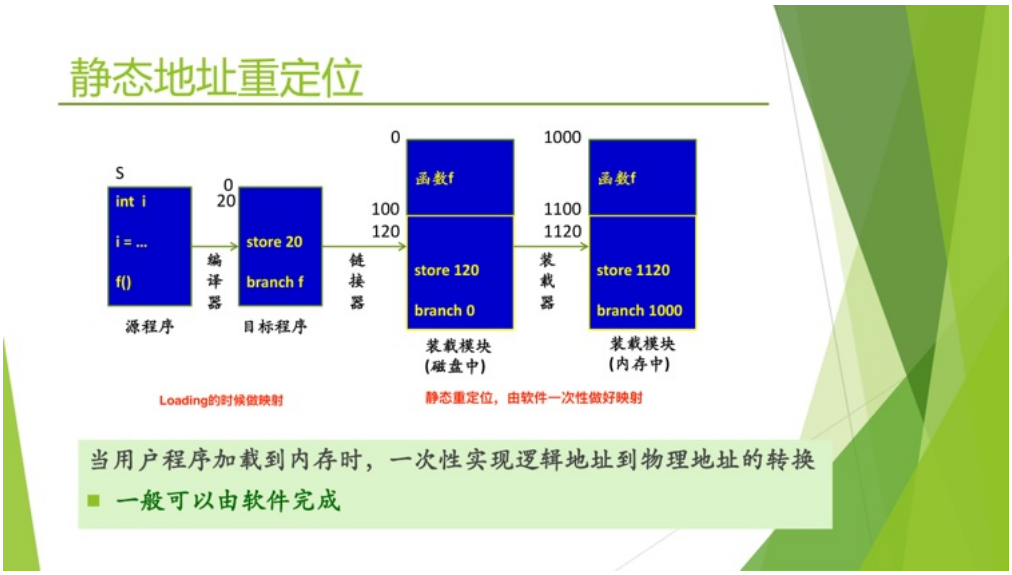
- **逻辑地址（虚拟地址）**：编译后的目标代码采用相对地址；不能直接用于内存访问；
- **物理地址**：内存中实际存储单元地址；
- **地址重定位**：把逻辑地址转换为运行时可直接寻址的物理地址；
- 目的是让 CPU 执行指令时正确访问内存单元。

5.5.3 静态地址重定位

- 装载时由软件一次性完成全部地址转换；
- 简单，但灵活性差：装载完后程序不能再移动；
- 当前主要用于一些嵌入式系统。



图示：逻辑地址来自编译/汇编后的目标代码，相对于首地址 0 编址；物理地址是内存中存储单元的地址，可被 CPU 直接寻址。



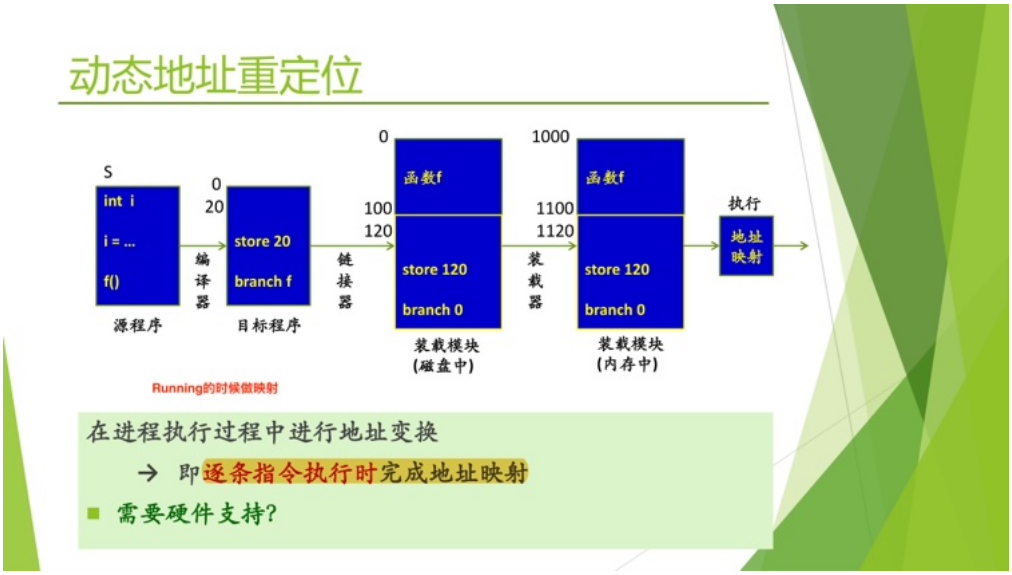
图示：装载时一次性将整个模块的逻辑地址映射成物理地址，不需硬件支持。

5.5.4 动态地址重定位与 MMU

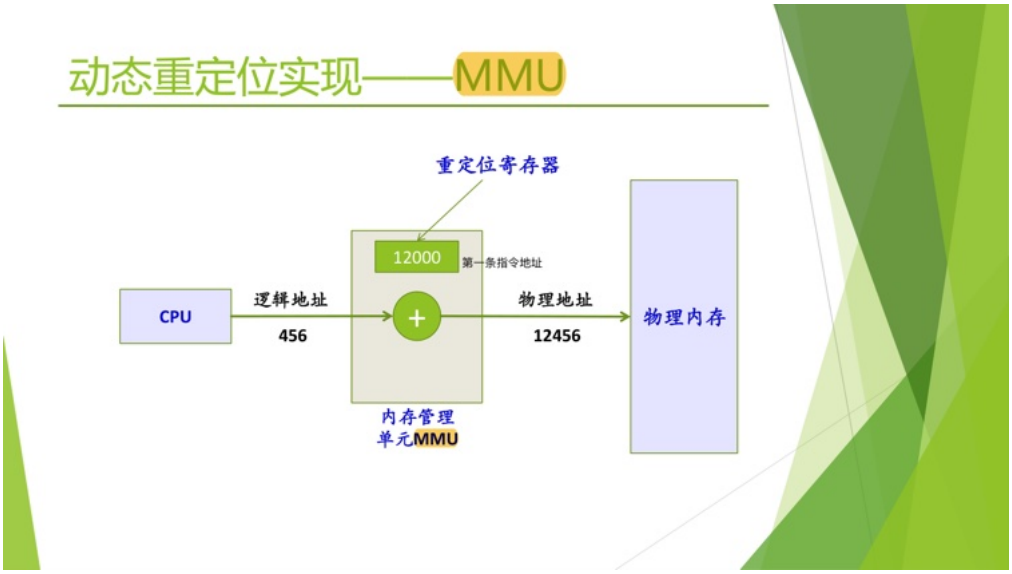
- 进程执行时逐条指令动态地址变换；
- 需要硬件 **MMU**（Memory Management Unit）支持；
- 重定位寄存器、页表寄存器（如 x86 CR3、RISC-V `satp`）保存当前进程的映射信息；
- 是现代 OS 的主流方案。

5.5.5 地址保护

- 通过基址 + 界限寄存器检查每次访问是否落在合法范围内；
- 越界则触发硬件异常，进入 OS；
- 进入临界区时由 OS 通过特权指令加载这两个寄存器；



图示：在程序执行时，每条指令运行时都做地址映射；需要硬件支持。



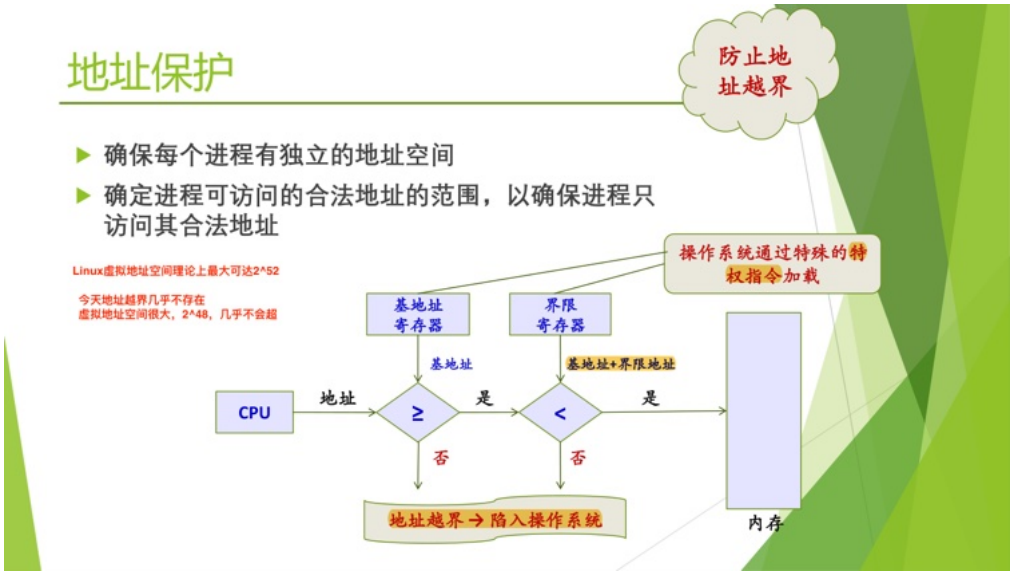
图示：CPU 把逻辑地址送入 MMU，MMU 加上重定位寄存器，得到物理地址。

- 在虚拟地址空间巨大（如 2^48）时，地址越界几乎不存在，但保护机制仍然保留。

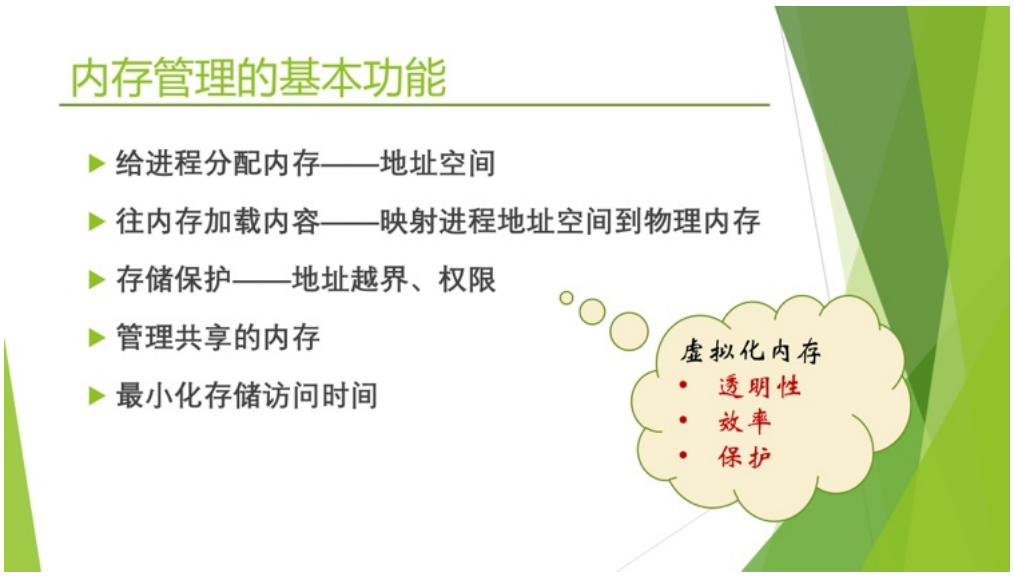
5.5.6 内存管理的基本功能

总结内存管理要做的事：

- 给进程分配地址空间；
- 把内容映射到物理内存；
- 存储保护（地址越界、权限）；
- 管理共享内存；
- 最小化访问时间；
- 虚拟化内存：要求透明性、效率、保护。



图示：基地址寄存器 + 界限寄存器 + 比较器；越界则陷入操作系统。



图示：分配地址空间、加载内容、保护、共享、最小化访问时间、虚拟化内存（透明性 / 效率 / 保护）。

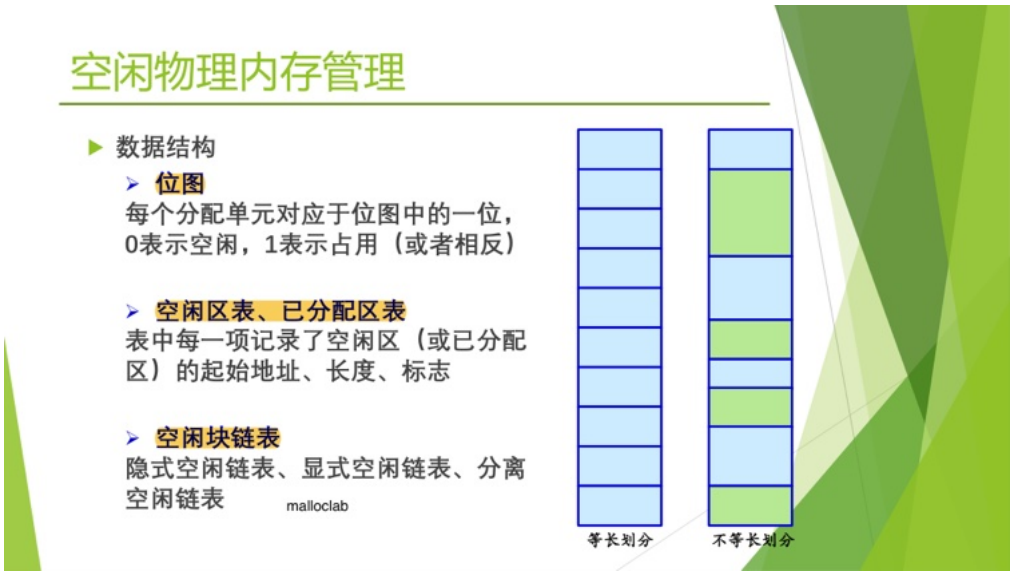
5.6 物理内存管理

5.6.1 空闲物理内存的数据结构

- **位图 (Bitmap)**：把内存按固定大小切成分配单元，每个单元在位图中占 1 位（0 空闲，1 占用）。等长划分。
- **空闲区表 / 已分配区表**：表项记录起始地址、长度、标志。
- **空闲块链表**：
  - **隐式空闲链表**：每块通过头部 / 尾部信息隐式串起来；
  - **显式空闲链表**：用指针串起所有空闲块；
  - **分离空闲链表 (segregated list)**：按大小分类的多条链表，是 Linux 伙伴系统、`malloc`



图示：本节包括位图法、空闲区表、空闲区链表，以及首次 / 最佳 / 最差适配等算法。



图示：位图、空闲 / 已分配区表、空闲块链表（隐式 / 显式 / 分离）。

实现的常用结构。

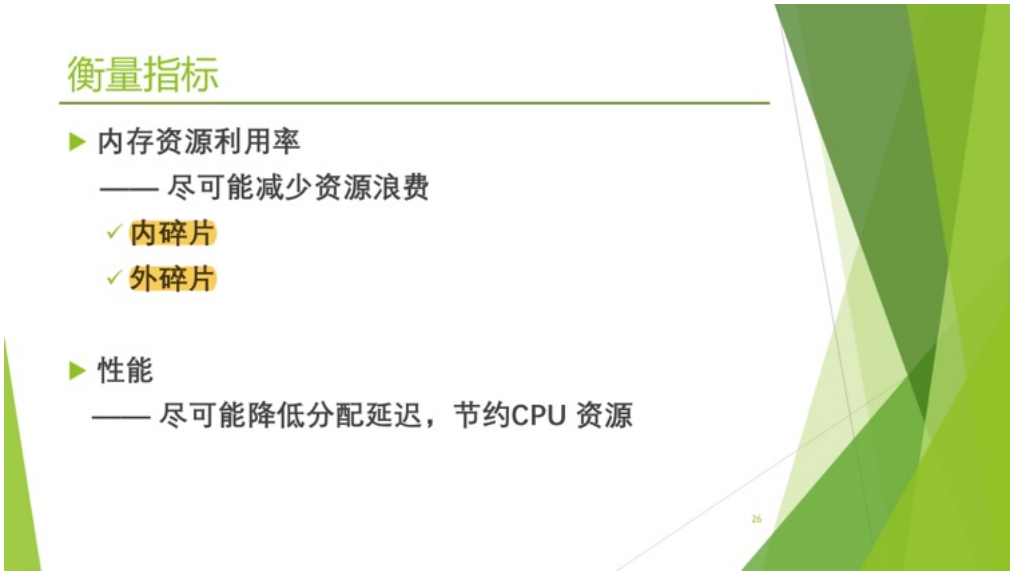
> **小练：** ICS 课程的 malloclab 中可以亲手实现这些结构，理解每种实现的复杂度与碎片表现。

5.6.2 衡量指标

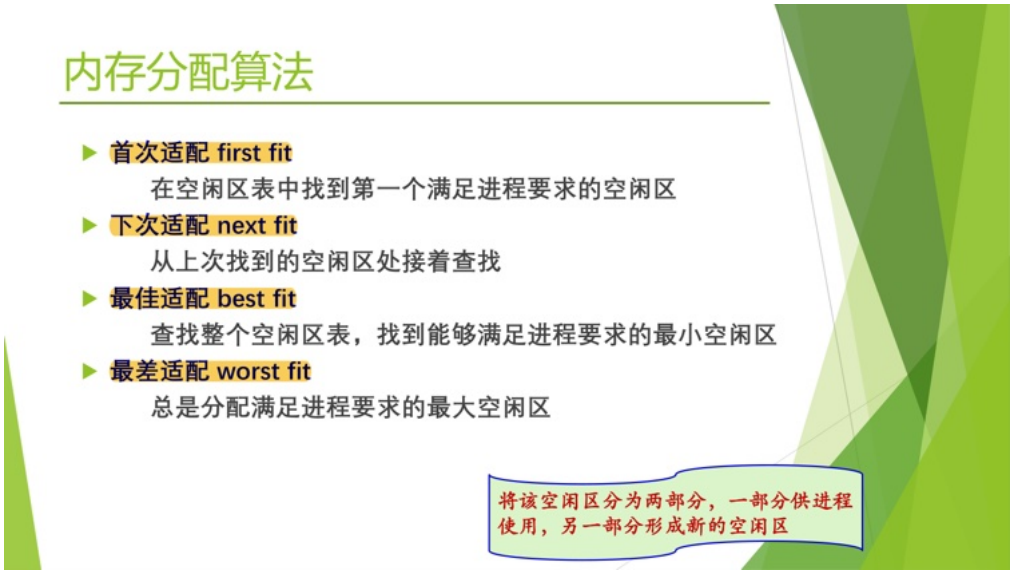
- **内碎片**：分配单位大于实际需求时，内部多出的部分；
- **外碎片**：空闲块过散，整体足够却无法满足某次分配；
- **性能**：分配 / 释放的时间复杂度、cache 友好程度等。

5.6.3 内存分配算法

- **首次适配（First Fit）**：从空闲区表头找到第一个满足要求的块；速度快；外碎片偏向头部。



图示：内存资源利用率（减少内 / 外碎片）+ 性能（降低分配延迟、节约 CPU）。



图示：首次适配、下次适配、最佳适配、最差适配。

- **下次适配（Next Fit）**：从上次找到的位置接着找；外碎片更分散；空间分布更均衡。
- **最佳适配（Best Fit）**：找最小满足要求的块；剩余空间最小；但产生大量难用的小碎片。
- **最差适配（Worst Fit）**：找最大块切；剩余空间较大易复用，但易破坏大块。
- 找到块后切两部分：一部分给进程，剩下作为新空闲区。

5.6.4 分配 / 回收举例

回收的核心是“**合并相邻空闲区**”，否则外碎片会迅速增加。

5.6.5 伙伴系统

- 思想：把内存看作一块大小为 2^U 的块；
- 分配时：



图示：4 个进程已经占据若干区域，空闲区表与已分配区表的内容。



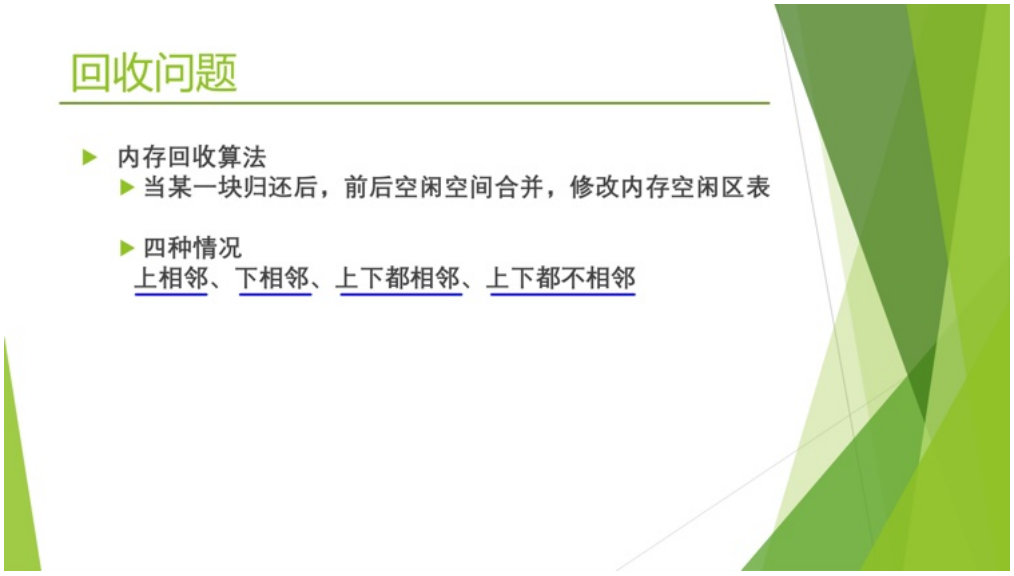
图示：在新进程到达后空闲区表如何被切割与更新。

  - 进程申请 s 字节；
  - 若 2^(U-1) < s <= 2^U，分配整块；
  - 否则把当前块切成两个大小相等的伙伴，继续切到刚好满足 s 的最小块；
- 释放时：若两个伙伴块都空闲且来自同一次划分，则合并；
- 优点：实现简单，分配与释放是 O(log N)；与位运算配合极快；
- 缺点：内碎片可能高（按 2 幂分配）。

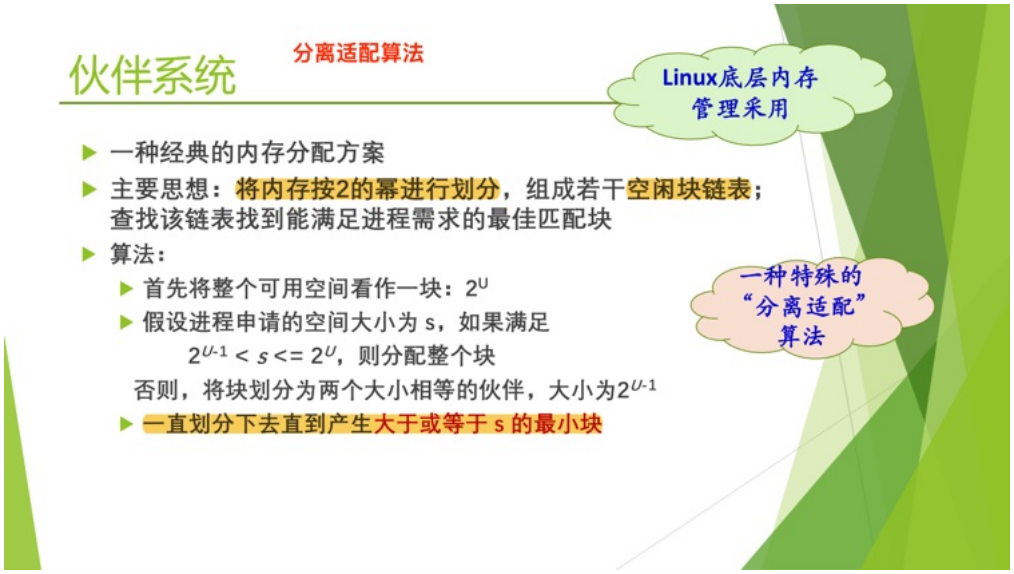
例子：

整个过程的关键是“**只有同源伙伴才能合并**”。

5.6.6 SLAB / SLUB / SLOB



图示：归还块后前后空闲区合并；四种情况：上邻、下邻、上下都邻、上下都不邻。



图示：一种经典的分离适配算法；按 2 的幂划分内存。

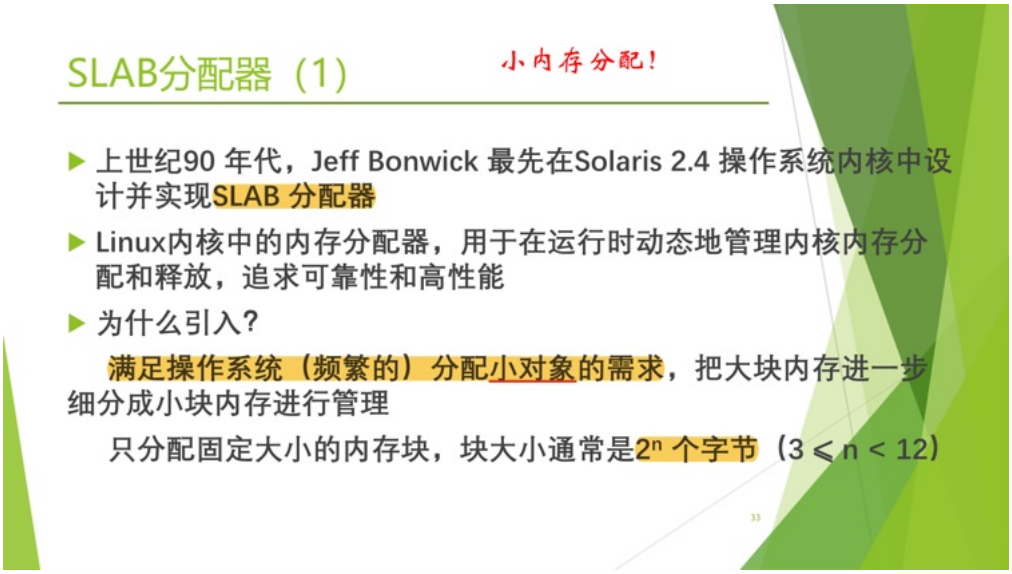
SLAB

- 解决“伙伴系统按页分配，内核要分配几十字节小对象，浪费大”的问题；
- 只分配固定大小的内存块，块大小通常是 2^n 字节（3 <= n < 12）。

- 把可用内存按大小划分为不同区域；
- 每个区域有一个空闲对象列表；
- 分配时在合适列表中找一个空闲对象返回；
- 释放时优先放回本地高速缓存；
- 广泛应用于 Linux 文件系统、网络协议栈等模块；
- 可以通过 `cat /proc/slabinfo` 查看当前 slab 缓存。



图示：在 1MB 内存中陆续分配 / 释放 A、B、C、D、E 的过程。



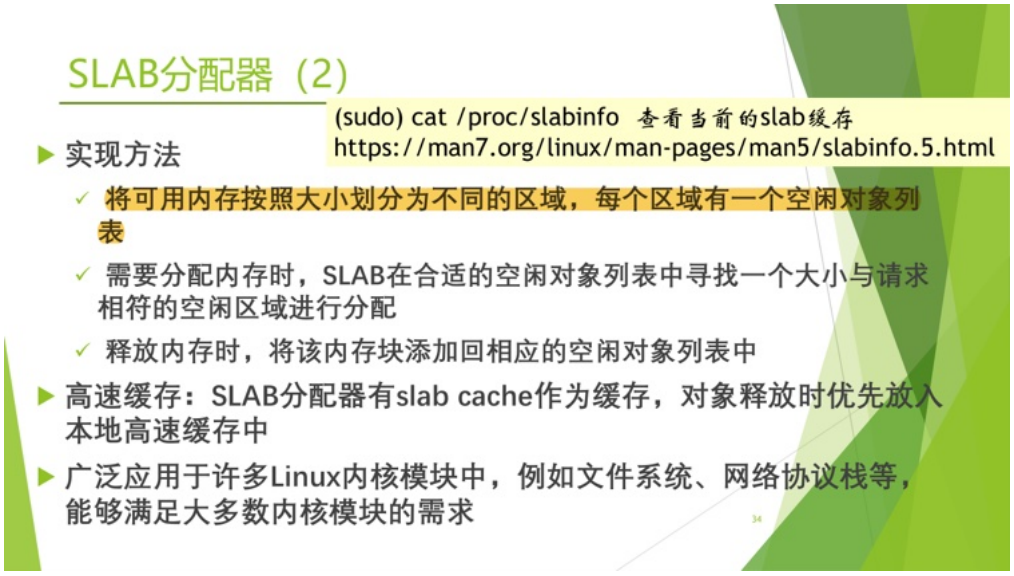
图示：1990 年代由 Jeff Bonwick 在 Solaris 中提出，专门用于内核小对象分配。

SLUB

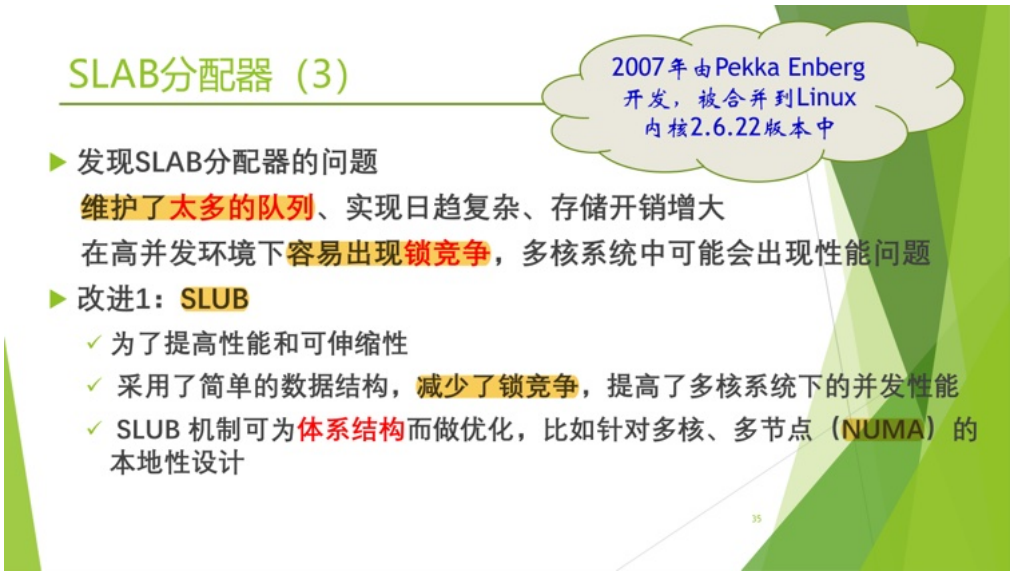
- SLAB 的问题：维护太多队列、复杂、锁竞争多、多核可扩展性差；
- SLUB 改进：
  - 数据结构更简单；
  - 减少锁竞争；
  - 针对多核 / NUMA 优化本地性；
- 现代 Linux 主线默认使用 SLUB。

SLOB

- 设计目标：尽可能简单、紧凑；
- 适用于小型嵌入式系统、资源受限的环境；



图示：每种大小一个空闲对象列表；带本地高速缓存（slab cache）。

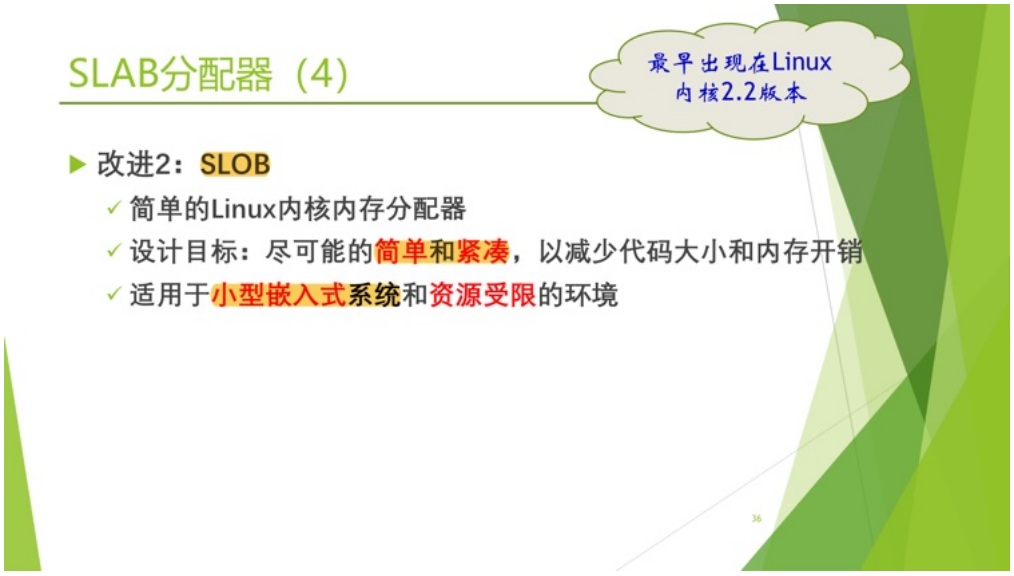


图示：2007 年 Pekka Enberg 提出，合入 Linux 2.6.22。

- 代码量小、常驻内存小，但性能与可扩展性不强。

> **总结：**伙伴系统 + SLAB/SLUB 构成了 Linux 内核两层物理内存管理：底层按页（4KB）分配；上层针对小对象做精细化管理。

5.7 基本内存管理方案



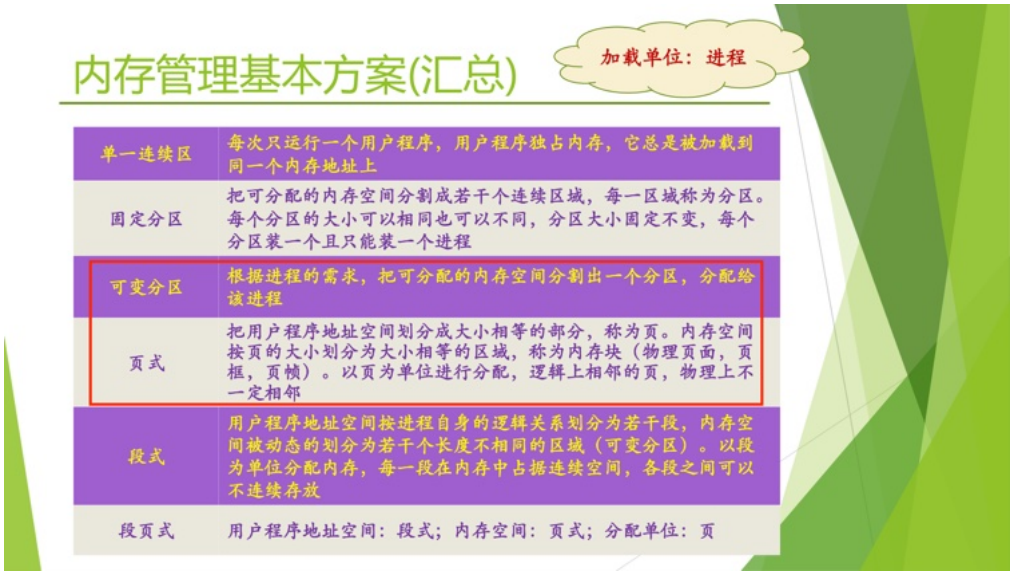
图示：最早出现在 Linux 2.2，用于嵌入式 / 资源受限系统。



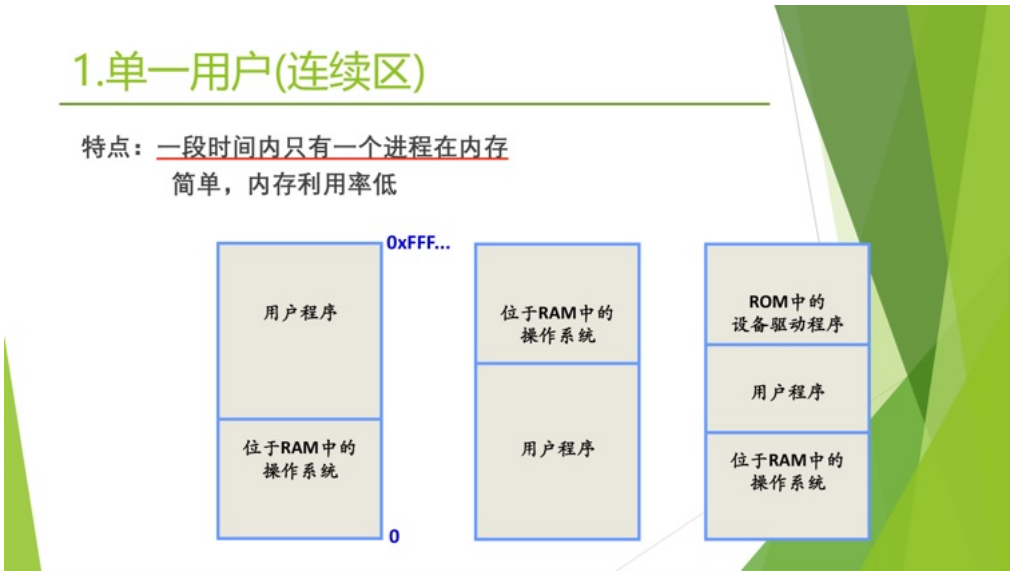
图示：单一连续区、固定分区、可变分区、分页、分段、段页式六种方案概览。

5.7.1 各方案对比

方案	用户地址空间	内存空间划分	分配规则
单一连续区	整体一段	整体可用	一次只放一个程序
固定分区	整体一段	预先切成若干固定分区	每分区放一个进程
可变分区	整体一段	按进程需要切出	按需分配
页式	等长“页”划分	等长“页框”划分	按页分配，逻辑相邻物理可不连续
段式	按程序逻辑分段	不等长可变分区	按段分配，每段连续，不同段可不连续
段页式	段式划分	页式划分	段式 + 页式结合



图示：六种方案的“用户地址空间划分 + 内存空间划分 + 分配规则”对比。



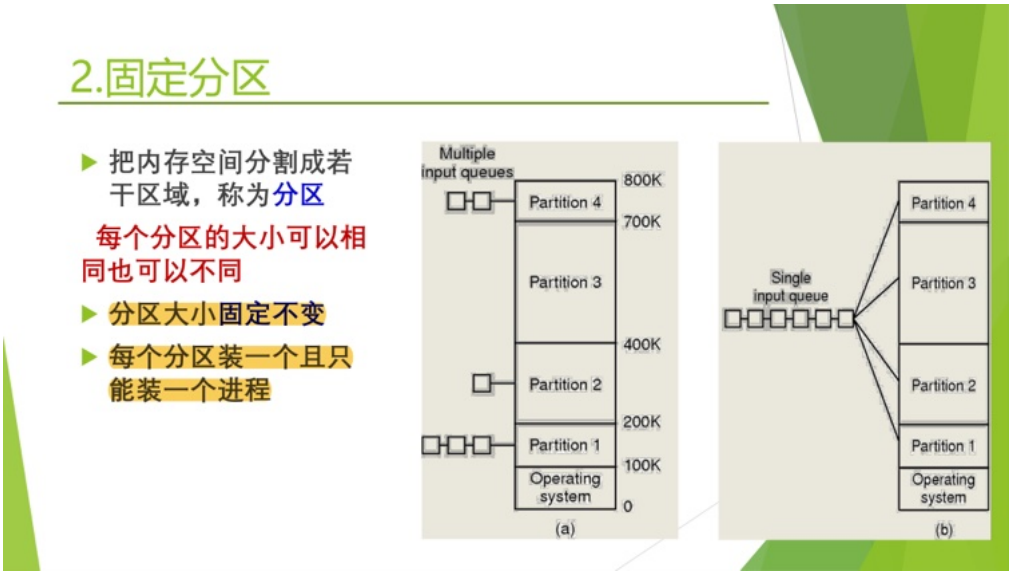
图示：内存中操作系统 + 用户程序，一次只跑一个进程。

5.7.2 单一连续区

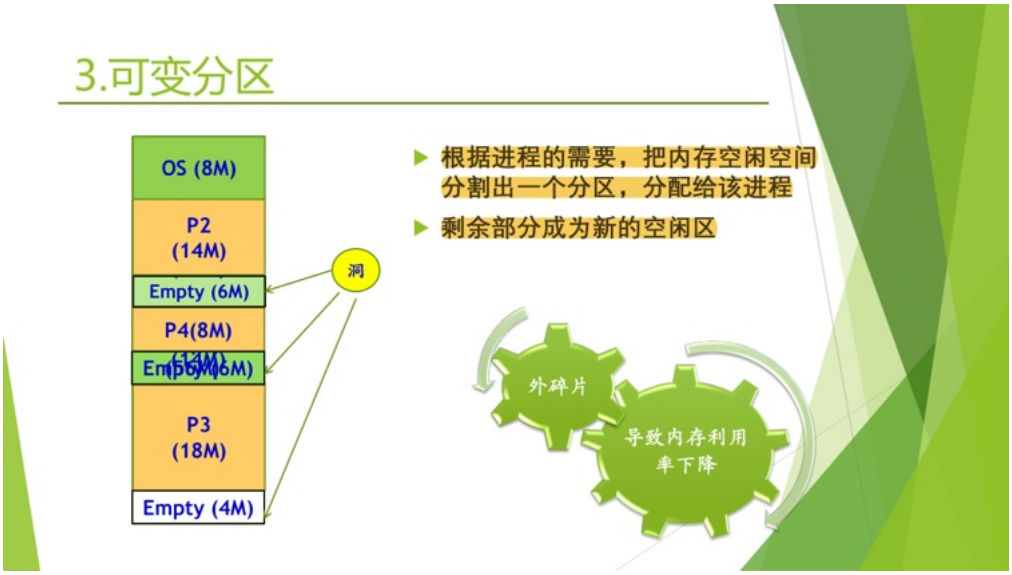
- 一段时间内只有一个进程在内存；
- 简单但内存利用率低；
- 早期 / 极简 OS 使用。

5.7.3 固定分区

- 分区数量、大小预先确定；
- 简单但灵活性差，内碎片显著。



图示：把内存切成若干固定分区，每分区放一个进程。



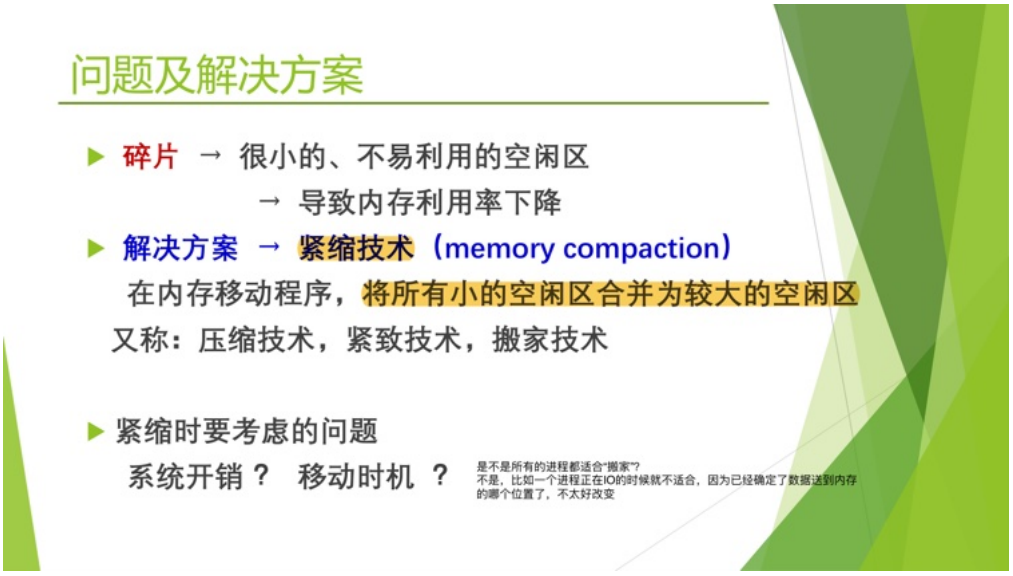
图示：按进程需要切出分区；剩余部分形成新空闲区；产生外碎片。

5.7.4 可变分区

- 按需分配；
- 产生外碎片（“洞”）；
- 需要 §5.3 的分配算法和 §6.4.1 的紧凑技术配合。

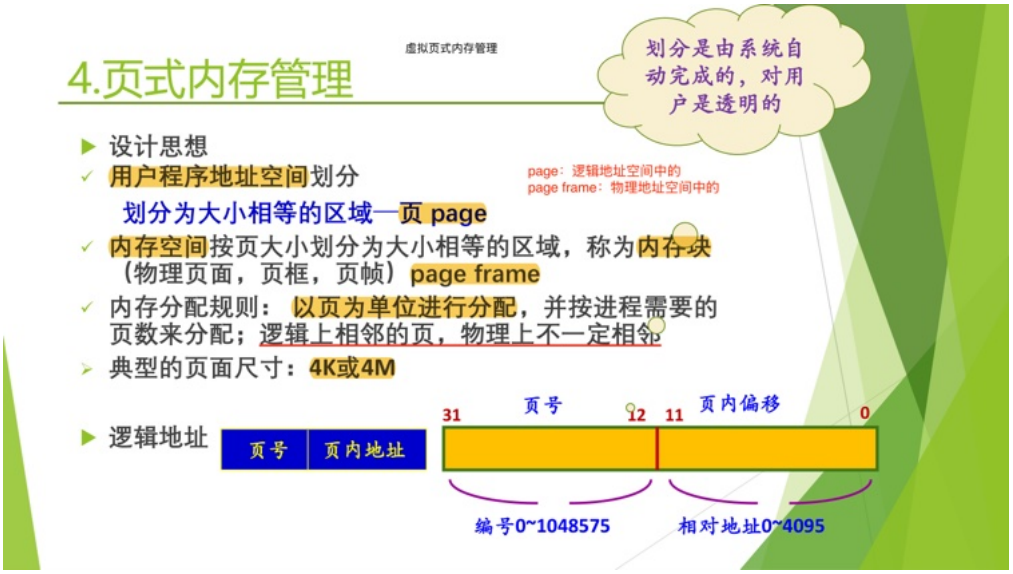
碎片与紧凑技术

- **碎片**：很小、不易利用的空闲区，导致利用率下降；
- **紧凑技术（compaction）**：移动程序，把所有小空闲区合并成大块；
- 需要硬件支持动态地址重定位才好做（否则要重写程序中的所有地址）；
- 不是所有进程都适合搬家：例如正在 I/O 的进程，DMA 已经把目标地址固定，不便搬动。



图示：通过移动程序将小空闲区合并；要考虑系统开销、移动时机以及哪些进程可被搬家。

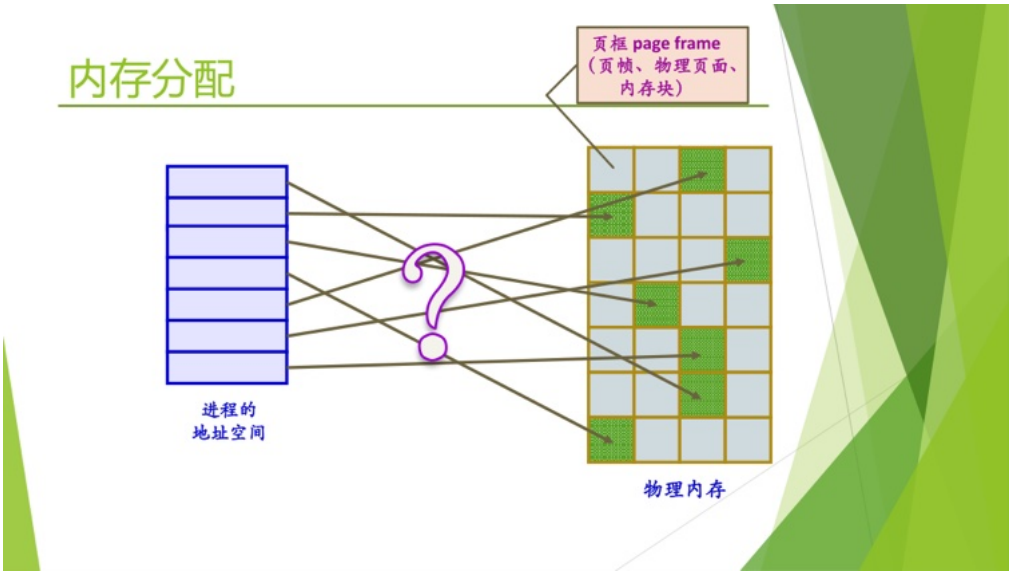
5.7.5 页式内存管理



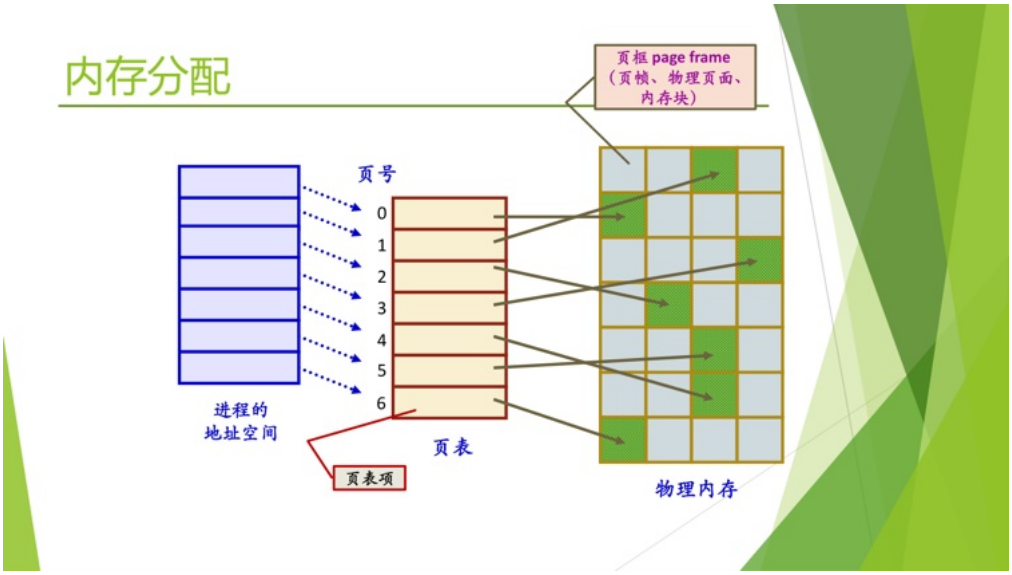
图示：用户地址空间划分为等长“page”；内存空间划分为等长“page frame”。

- 把用户地址空间划分为大小相等的 page；
- 内存空间划分为大小相等的 page frame（页框、物理页面、内存块）；
- 以页为单位分配；逻辑相邻的页在物理上不一定相邻；
- 典型页大小 4KB 或 4MB；
- 划分由系统自动完成，对用户透明；
- 逻辑地址 = 页号 + 页内偏移（如 32 位下，页号 0–11 位、页内偏移 12–31 位）。

内存分配视图



图示：进程地址空间和物理内存中页框的映射。



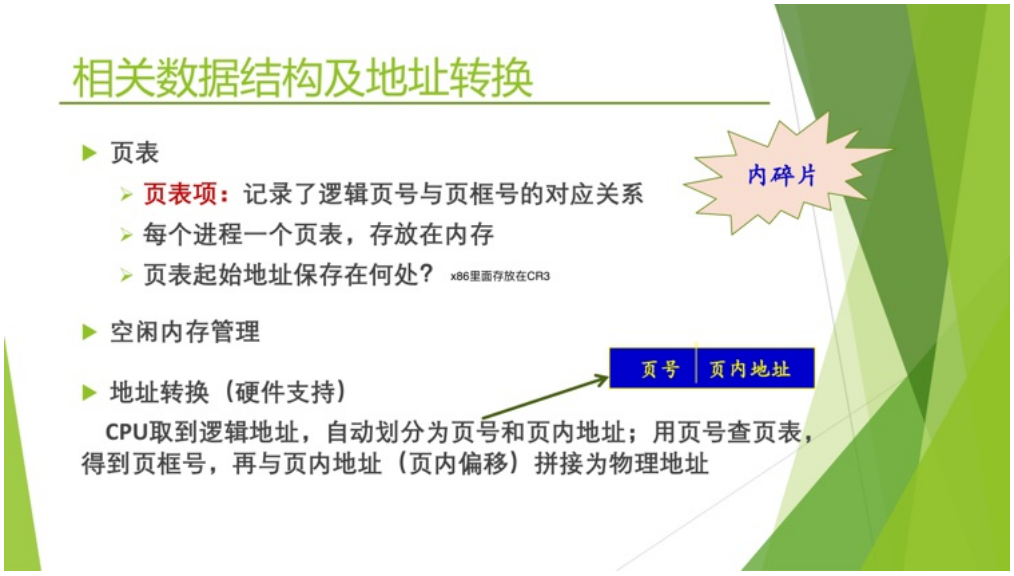
图示：页表把每个逻辑页号映射到对应的物理页框号。

数据结构与地址转换

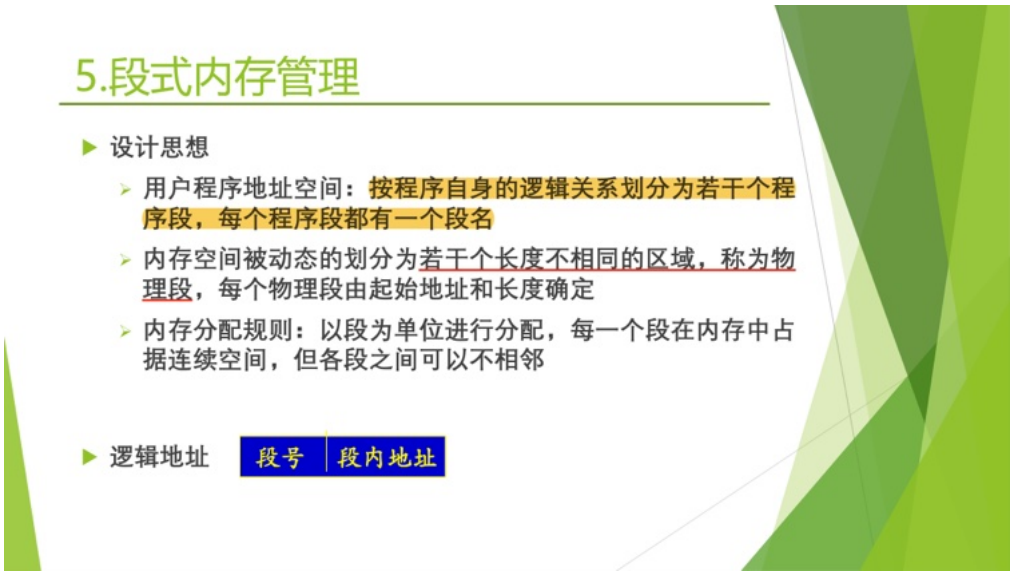
- **页表**：每个进程一张，存放在内存；
- **页表项**：记录逻辑页号 -> 物理页框号；
- **页表起始地址**：保存在专用寄存器（x86 的 CR3、RISC-V 的 `satp`）；
- 地址转换：CPU -> 逻辑地址 -> 页号 -> 查页表 -> 页框号 -> 拼接页内偏移 -> 物理地址；
- **碎片**：页式以页为单位分配，存在内碎片，但没有外碎片。

5.7.6 段式内存管理

- 用户地址空间按程序自身的逻辑关系划分若干程序段，每段有段名；
- 内存空间动态划分为若干长度不同的物理段；



图示：CPU 取到逻辑地址 -> 划分为页号 / 页内偏移 -> 用页号查页表 -> 拼接为物理地址。



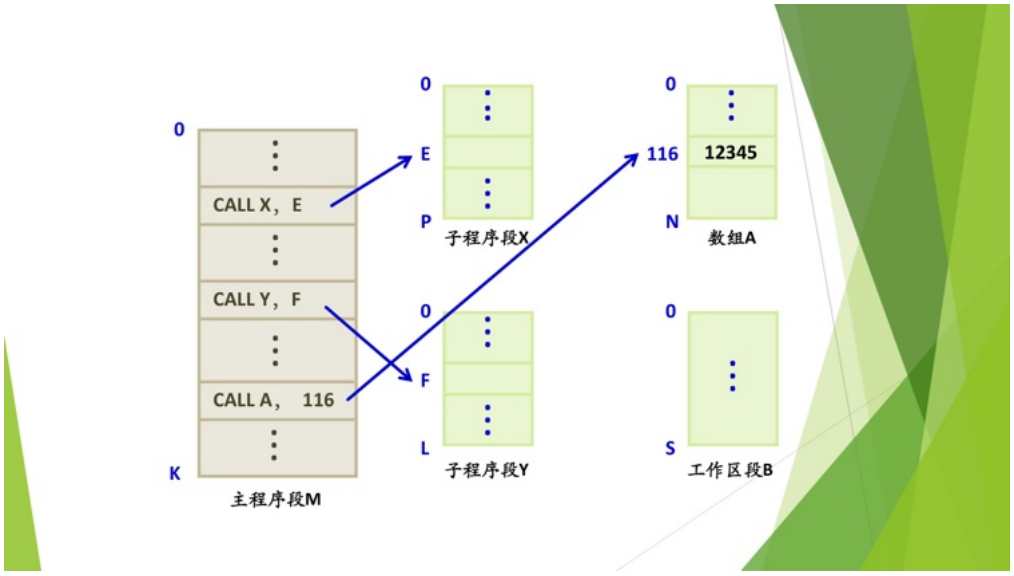
图示：用户地址空间按程序逻辑分段，每段有名字；内存空间动态划分若干长度不等的物理段。

- 每个物理段由起始地址 + 长度确定；
- 以段为单位分配，每段在内存中连续，但段之间可以不相邻；
- 逻辑地址 = 段号 + 段内地址。

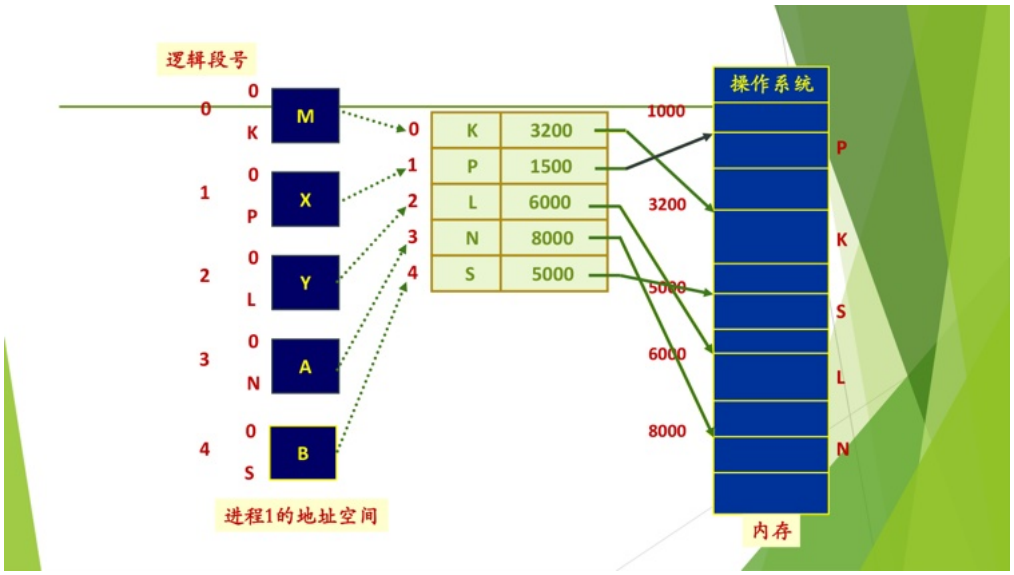
段式示意

数据结构与地址转换

- 段表记录段号、段首地址、段长度；
- 物理内存管理同可变分区；
- 转换需要硬件支持；
- 优点：保留程序的逻辑结构，便于共享和保护；



图示：主程序段 M、工作区段 B、子程序段 X / Y、数据段 A 等，互相之间有 CALL 关系。



图示：进程 1 的地址空间中各段被映射到内存中具体的起始地址；段表记录段号 -> 段地址 + 长度。

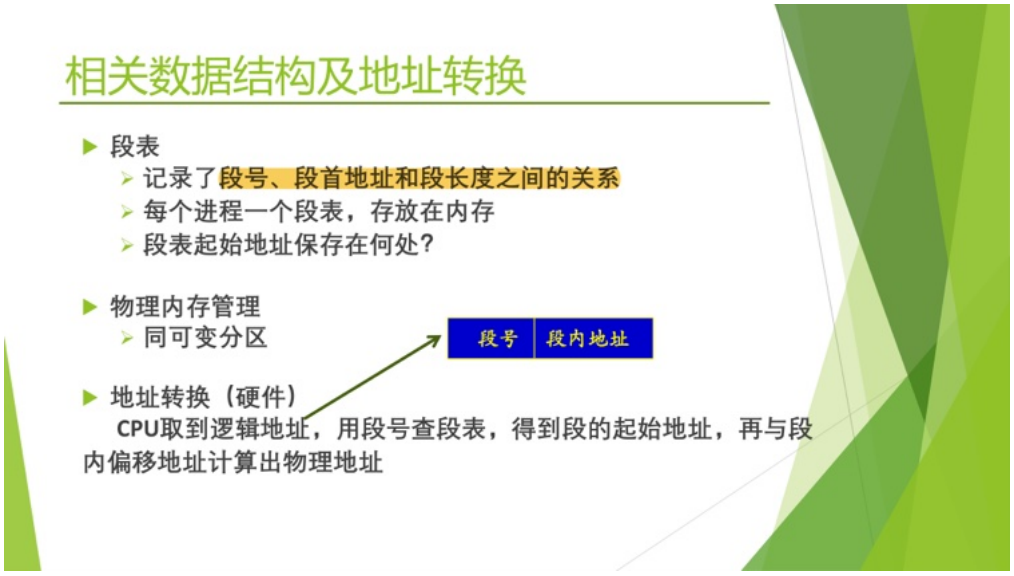
- 缺点：与可变分区一样有外碎片。

5.7.7 段页式内存管理

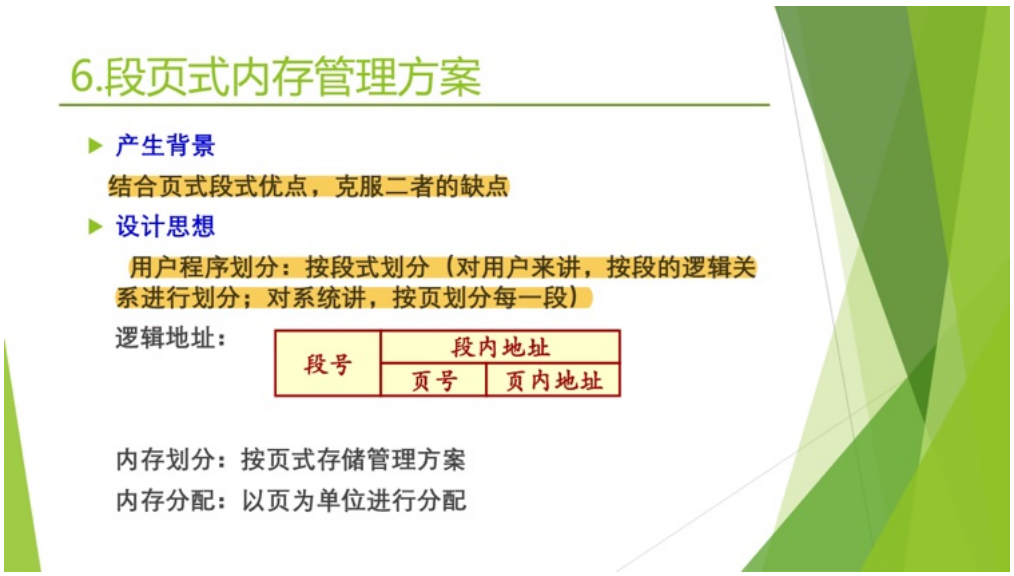
- 结合页式与段式优点；
- 用户视角：按段；系统视角：按页；
- 内存按页分配；
- 逻辑地址 = 段号 + 段内地址（段内地址 = 页号 + 页内地址）。

数据结构与地址转换

- 一个进程：一张段表 + 多张页表（每段一张）；
- 段表记录每一段的页表始址和页表长度；



图示：每个进程一张段表；CPU 取到逻辑地址 -> 段号 -> 查段表 -> 段起始地址 + 段内偏移 -> 物理地址。



图示：用户地址空间按段式划分，内存按页式管理；逻辑地址 = 段号 + 段内地址（段内地址再细分为页号 + 页内地址）。

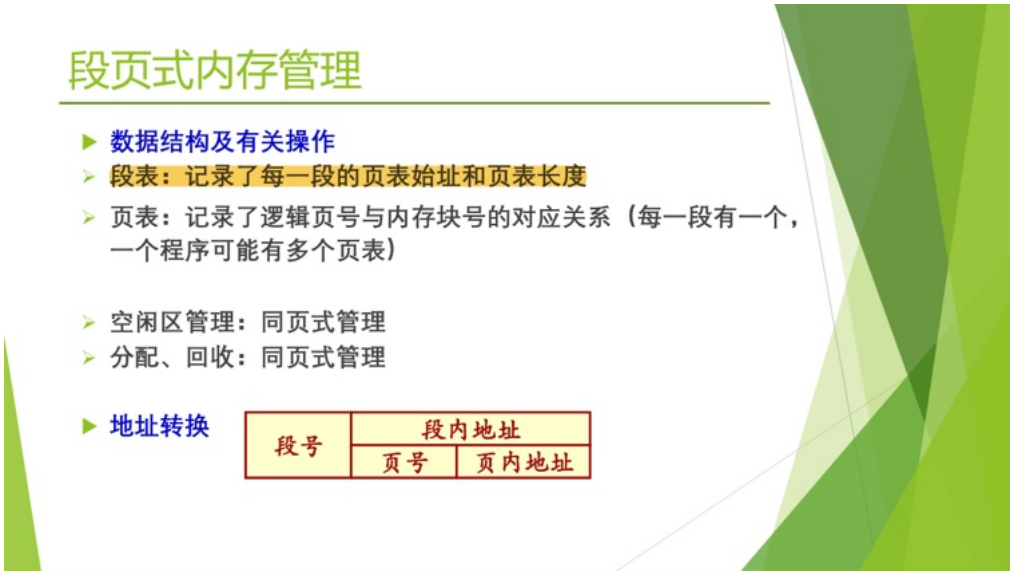
- 地址转换：段号 -> 段表 -> 页表始址 -> 页号 -> 页表项 -> 物理页框号 -> 拼接页内偏移；
- 实际硬件如 x86 段页机制是这种结构的简化。

5.8 内存“扩充”技术

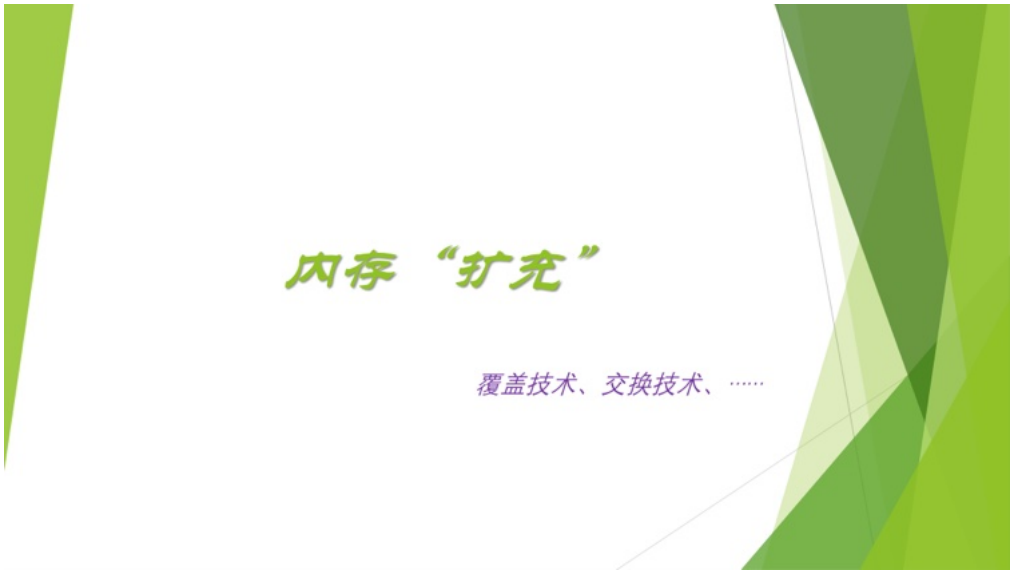
“扩充”不是物理上扩大内存，而是让有限内存运行更大或更多的程序。

5.8.1 覆盖技术 Overlaying

- 解决“程序大小超过物理内存总和”的问题；
- 在程序执行过程中，不同部分相互替代；
- 把不会同时执行的模块共享同一块内存；



图示：段表记录每段的页表起始地址 + 页表长度；每段对应一张页表；空闲管理 / 分配 / 回收同页式。



图示：覆盖、交换技术等内存扩充手段。

- 程序员声明覆盖结构，由 OS 完成自动覆盖。

例子

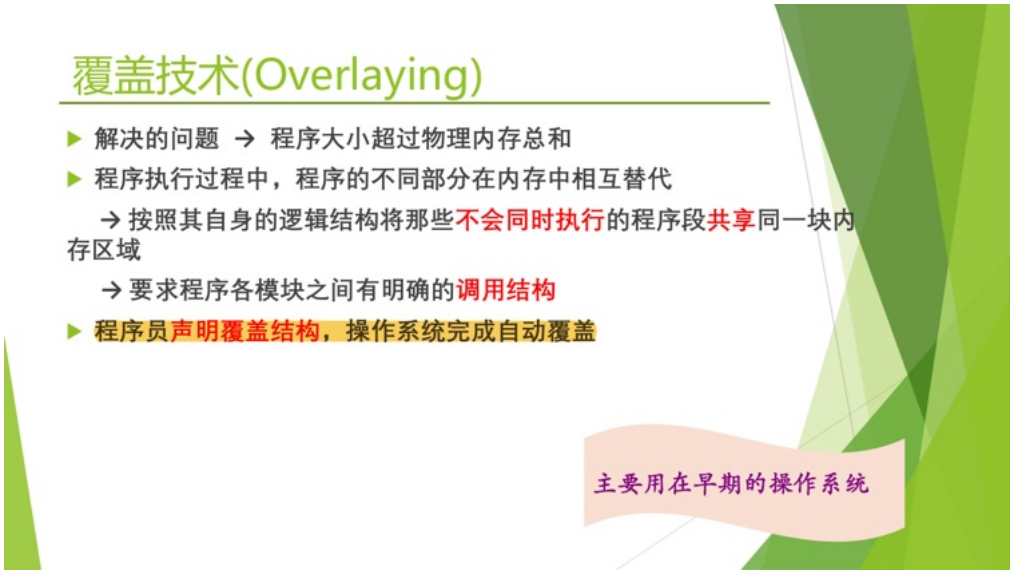
- 程序 X 的常驻区与几个互斥的覆盖区放在同一块内存；
- 通过覆盖让总占用从 54K 降到 30K。

缺点

- 增加编程难度：用户必须自己设计覆盖结构；
- 增加执行时间：需要从外存装入覆盖模块；
- 是一种“**用时间换空间**”的策略；
- 主要在早期 OS 中使用。



图示：紧凑、覆盖、交换、虚存四类技术。



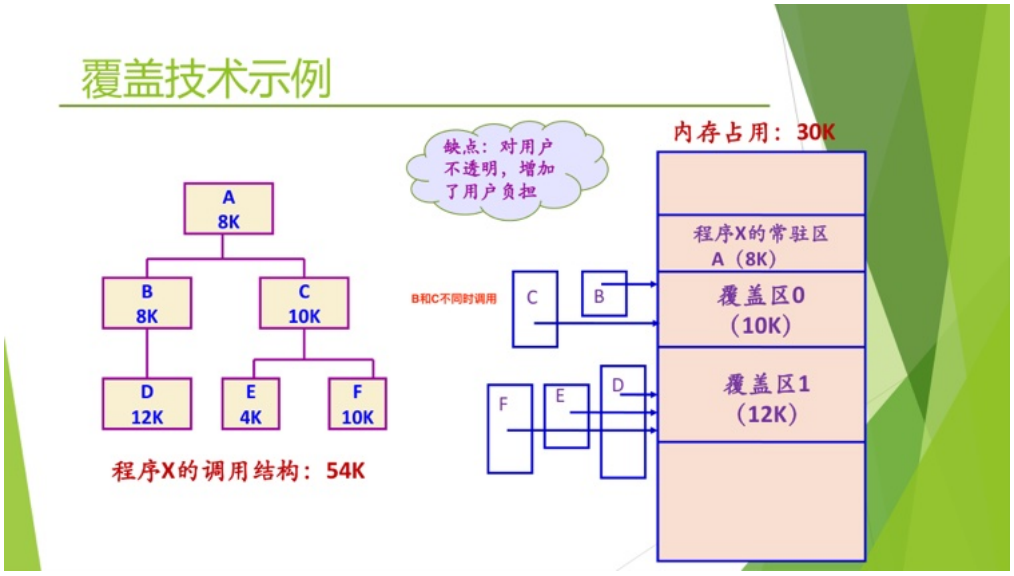
图示：把不会同时运行的模块共享同一块内存区域；要求模块之间有清晰调用关系。

5.8.2 交换技术 Swapping

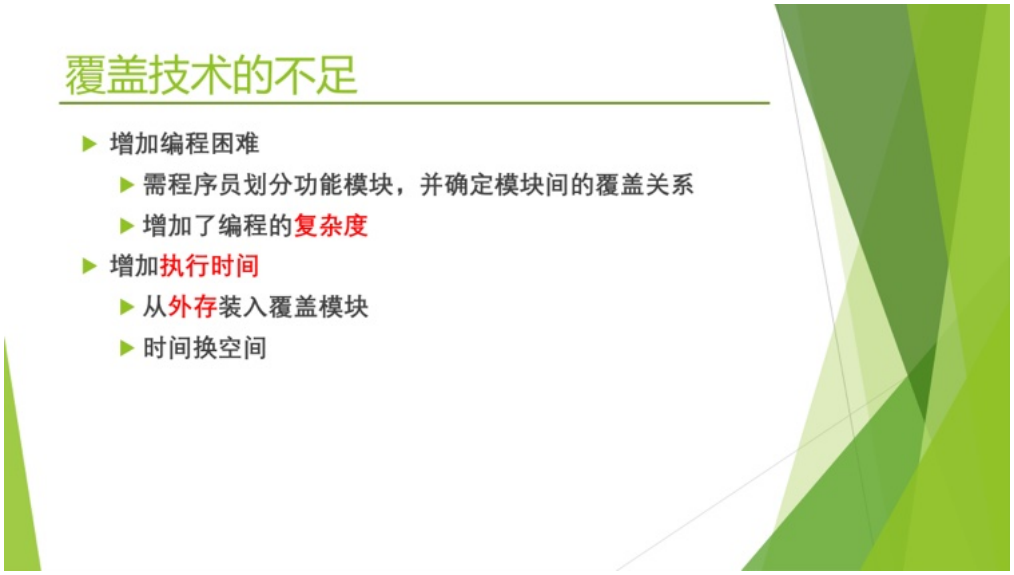
- 早期用于小型分时系统；
- 设计思想：内存空间紧张时，把一些进程暂时移到外存；把外存中要执行的进程换进来；
- “**进程在内存与外存之间动态调度**”。

实现要点

- **进程哪些部分需要换出？**主要是运行时创建或修改的内容（栈与堆）；磁盘上原有的代码 / 数据不必交换；
- **交换区**：一般指定一块特殊磁盘区域作为 swap space，连续的磁道，OS 用底层磁盘读写高效访问；
- **何时换？**不用就换出（早期用法）；或在内存空间不够 / 有不够危险时换出（与调度器结合）；



图示：程序 X 共需 54K，但只用 30K 内存运行（常驻区 8K + 覆盖区 0 10K + 覆盖区 1 12K）。

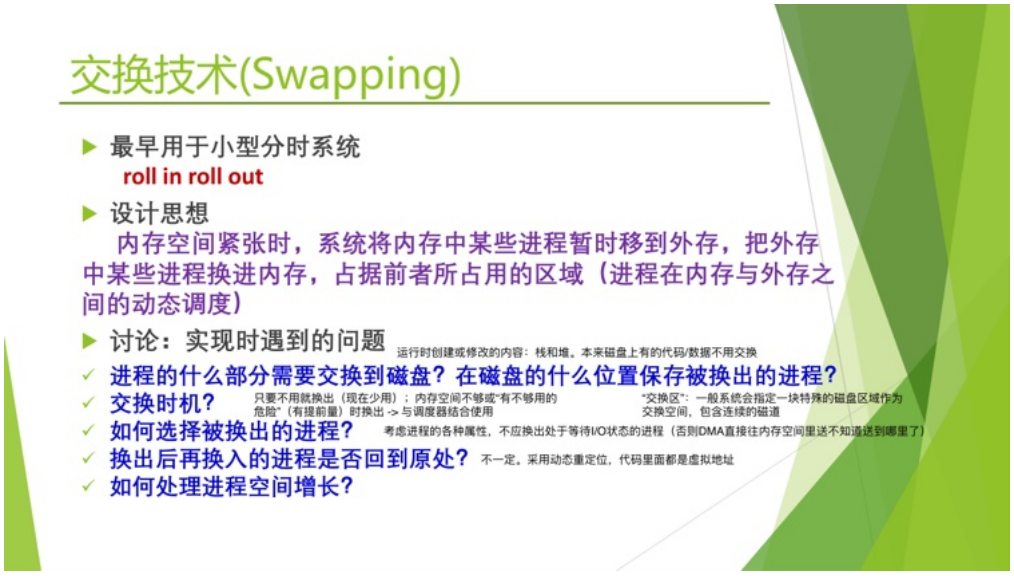


图示：增加编程难度（要划分模块、定义覆盖关系）；增加执行时间（从外存装入覆盖模块）。

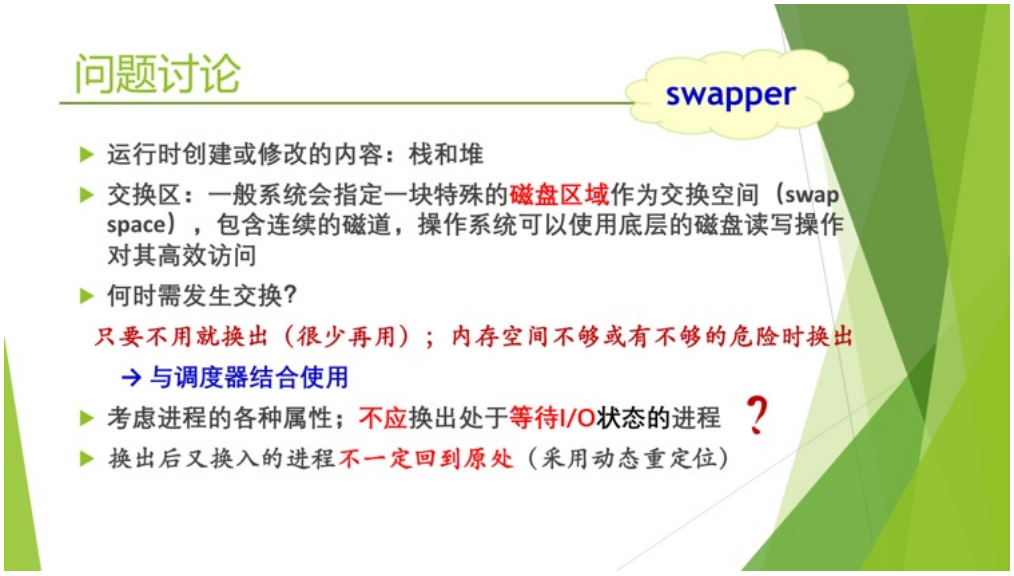
- **选谁换出？**考虑进程属性：不应换出处于等待 I/O 的进程，否则 DMA 仍在向原内存写入；
- **是否回原处？**不一定。借助动态重定位，换回的进程可以放在不同位置；
- **如何处理空间增长？**重新分配更大区域。

进程空间的增长

- 数据段、栈段会动态增长；
- 若换回时空间不够，要重新分配更大的区域；
- 这是后续虚拟内存机制要解决的核心问题之一。



图示：把内存中不活跃进程换到外存，把外存中需要的进程换入内存；roll in / roll out。

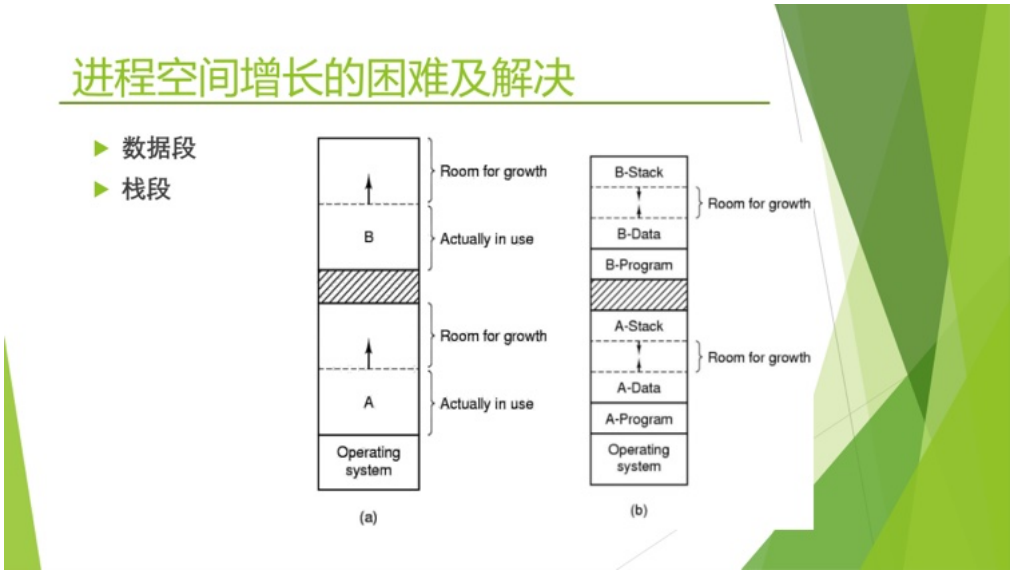


图示：交换区的位置、何时交换、如何选择被换出进程、是否回到原处、如何处理空间增长。

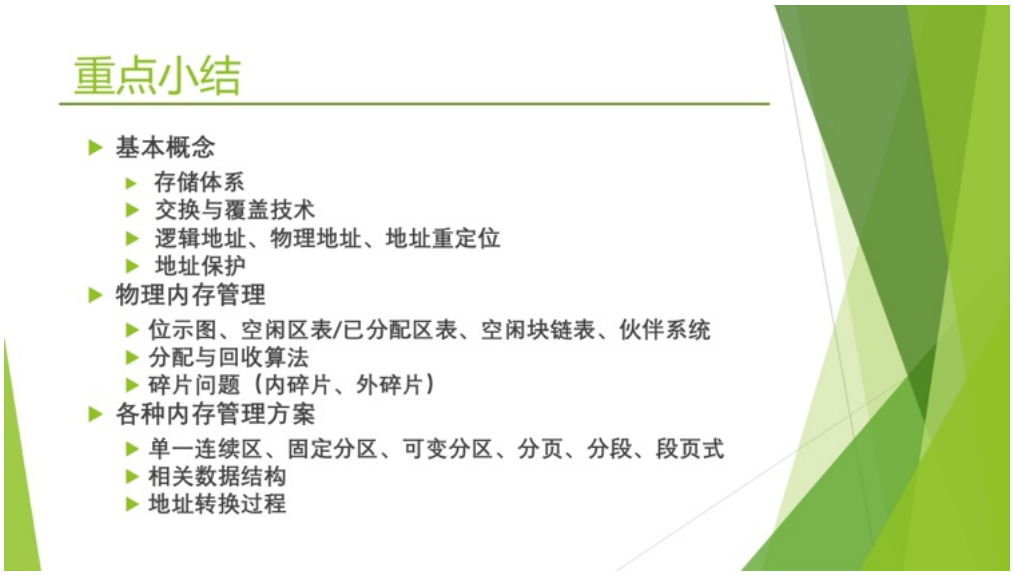
5.9 重点小结

记住几条主线：

- **基础概念**：存储体系、逻辑地址 / 物理地址、地址重定位、地址保护；
- **物理内存管理**：位图、空闲 / 已分配区表、空闲块链表、伙伴系统；分配与回收算法；内碎片 / 外碎片；
- **基本方案**：单一连续区、固定分区、可变分区、分页、分段、段页式；它们的数据结构和地址转换流程；
- **扩充技术**：紧凑、覆盖、交换；
- 这些是下一讲虚拟存储技术的基础——分页 + 交换的组合在虚存中演化为请求调页 + 页面置换。



图示：数据段与栈段在交换 / 重新分配时的处理。



图示：基础概念、物理内存管理、各种方案的相关数据结构与地址转换。

LECTURE 6

第 6 讲

虚拟存储技术

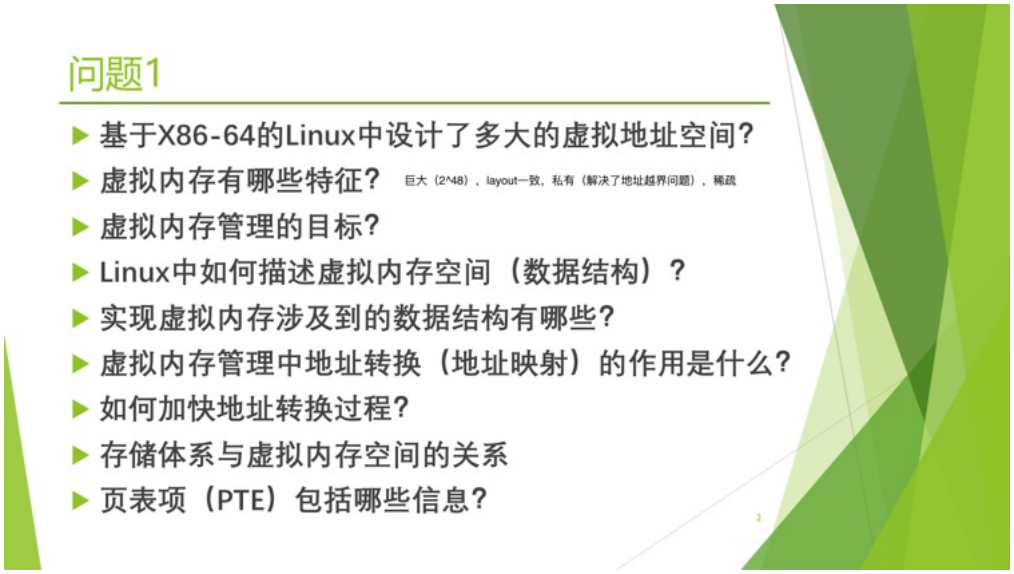
Operating Systems Course Notes

6.1 本讲导读

- 这一讲在第 5 讲的基础上回答两个核心问题：
  1. 内存有限、程序却很大，**怎么让程序看到一个比物理内存更大的地址空间？**
  2. 这个抽象在硬件、内核数据结构和算法层面具体怎么实现？
- 知识脉络：
  1. 虚拟页式存储管理的基本思想；
  2. 页表设计 / 多级页表 / 反置页表；
  3. 地址转换与 TLB；
  4. 缺页异常处理与驻留集管理；
  5. 各种页面置换算法及工作集模型；
  6. 清除策略、加载控制、抖动；
  7. 内存映射文件与 COW；
  8. Windows 虚拟存储管理。

> **关键提醒：**虚拟存储技术是“**操作系统资源转换技术**”的典型应用——用 CPU 时间和磁盘空间换取物理内存空间。

6.2 开篇问题

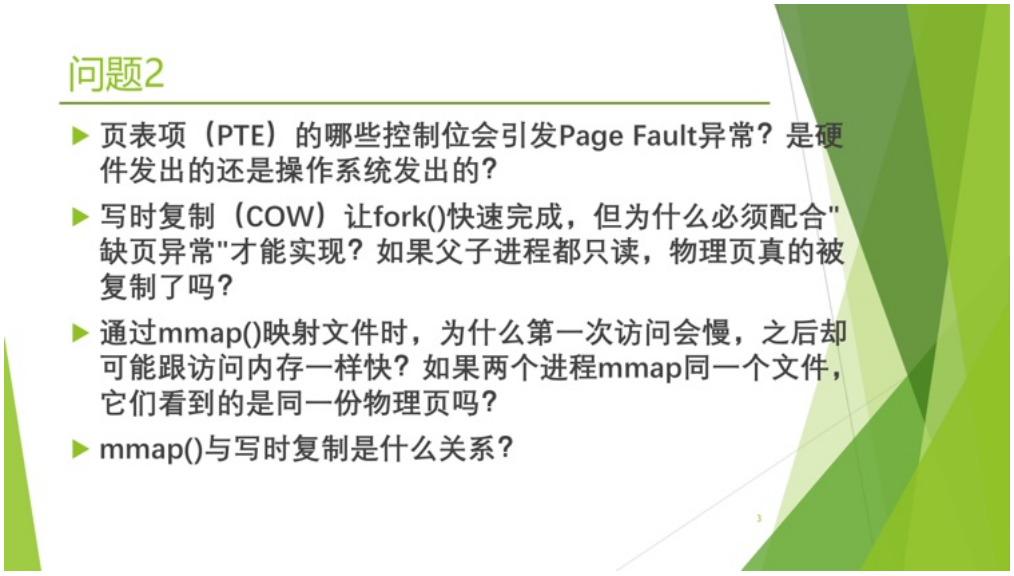


图示：x86-64 Linux 的虚拟地址空间大小、虚存特征、目标、数据结构、地址转换、TLB、PTE 内容等。

下面给出全部思考题的精炼解答。

> **问 1：** x86-64 Linux 设计了多大的虚拟地址空间?

解答：实际只用 48 位，即 2 256 TB。其中高 16 位必须是 47 位的符号扩展（“canonical 地



图示：PTE 哪些位会引发缺页异常、COW 与缺页的关系、mmap 与共享 / 写时复制的关系。

址”），把空间切成低半区（用户）和高半区（内核）。

> **问 2：** 虚拟内存有哪些特征?

解答： 巨大（2 ）、layout 一致（每个进程看到一致的虚拟空间）、私有（解决越界问题）、稀疏（实际使用的部分远小于总空间）。

> **问 3：** 虚拟内存管理的目标?

解答： 抽象（让进程看到独立的连续地址空间）、效率（用 TLB / 多级页表降低开销）、保护（按权限位防越权访问）、共享（不同进程间映射相同物理页）、扩容（用磁盘扩展物理内存）。

> **问 4：** Linux 中如何描述虚拟内存空间?

解答： 使用 `mm_struct` 描述整个地址空间，里面挂一组 `vm_area_struct`（VMA）描述各段连续的虚拟区域；通过页表完成虚拟到物理的映射。

> **问 5：** 实现虚拟内存涉及哪些数据结构?

解答： 页表（PDE / PTE）、TLB、`mm_struct` / `vm_area_struct`（Linux）/ VAD（Windows）、PFN 数据库（Windows）/ `struct page`（Linux）、空闲链表 / 修改链表 / 后备链表等。

> **问 6：** 地址转换的作用?

解答： 把虚拟地址映射到物理地址；同时承担权限检查（读 / 写 / 执行 / 用户 vs 内核）、缺页检测、共享与 COW 触发。

> **问 7：** 如何加快地址转换？

解答： 引入 TLB（Translation Look-aside Buffer）保存页表项的子集；按内容并行查找；多级页表 + 大页 + ASID/PCID 减少 TLB 失效。

> **问 8：** 存储体系与虚拟内存空间的关系？

解答： 虚存横跨“**内存 + 磁盘**”两层存储，操作系统协调它们的使用，让 CPU 看到一个“接近内存速度、容量很大”的地址空间。

> **问 9：** PTE 包括哪些信息？

解答： 物理页框号、有效位、访问位、修改位、保护位、用户 / 内核位、缓存控制位等。详见 §3.1。

> **问 10：** PTE 的哪些位会引发 Page Fault？是硬件还是 OS 发出的？

解答：

- 硬件检查 PTE 的 `Present`（有效 / 驻留）、`R/W`、`U/S`、`X/N` 等控制位；
- 任意一个位与当前访问语义不符就触发缺页异常；
- **异常本身由硬件发出**，但其语义由内核解释（不在内存？权限不符？COW？）并决定如何处理。

> **问 11：** COW 让 fork 快速完成，但为什么必须配合缺页异常？父子都只读时物理页会被复制吗？

解答：

- COW 在 `fork` 时把父子进程的页表项都改成只读，并标记 COW；
- 真正的复制发生在“**写**”动作触发缺页异常时；
- 如果父子都只读，硬件不会触发异常，物理页**不会**被复制——共享一直保持。

> **问 12：** mmap 第一次访问慢、之后却像访问内存一样快？两个进程 mmap 同一个文件看到的是同一份物理页吗？

解答：

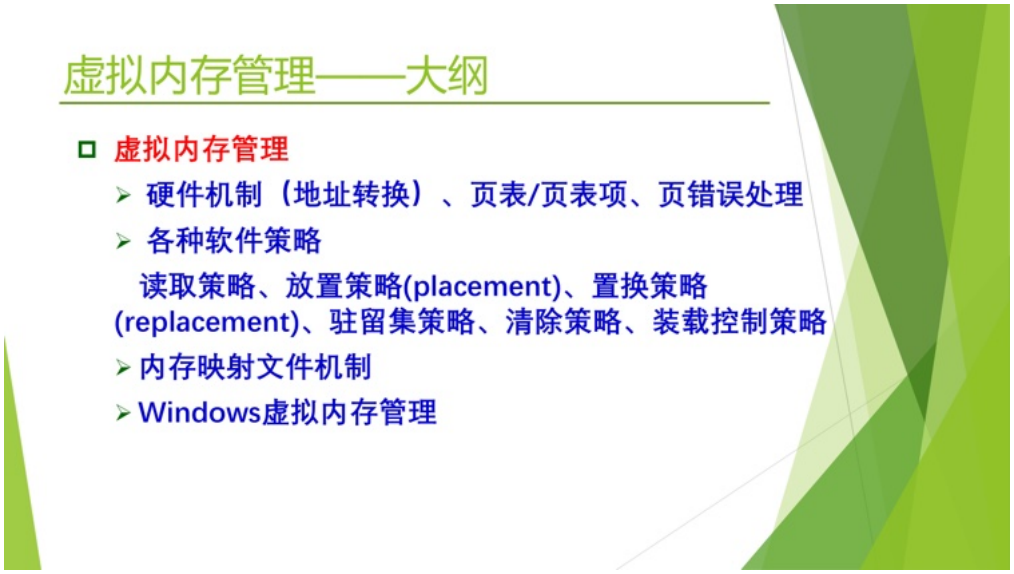
- 第一次访问：页表中尚未建立映射，触发缺页异常，由内核把磁盘块读入物理页；
- 之后访问：直接在内存中命中，速度等同于访问内存；
- 两个进程 mmap 同一个文件、并以共享方式映射时，看到的是**同一份物理页**——这就是 mmap 实现共享内存的方式。

> **问 13：** mmap 与 COW 的关系？

解答：

- 共享映射（`MAP_SHARED`）：多个进程共享同一物理页，写操作直接反映到文件和其他进程；
- 私有映射（`MAP_PRIVATE`）：以 COW 方式实现写隔离——读时共享，写时复制并断开共享；
- 因此 COW 是 mmap 私有映射的实现机制之一。

6.3 本讲大纲



图示：硬件机制（地址转换 / 页表 / 缺页处理）-> 软件策略（取页 / 放置 / 置换 / 驻留集 / 清除 / 装载控制）-> 内存映射文件 -> Windows 虚存管理。

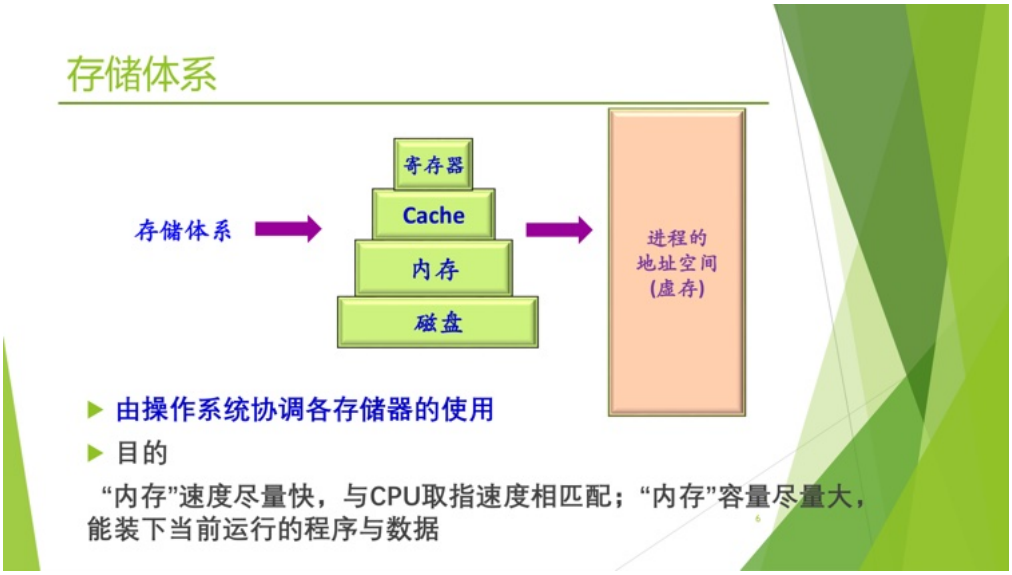
6.4 虚拟页式存储管理基础

6.4.1 存储体系

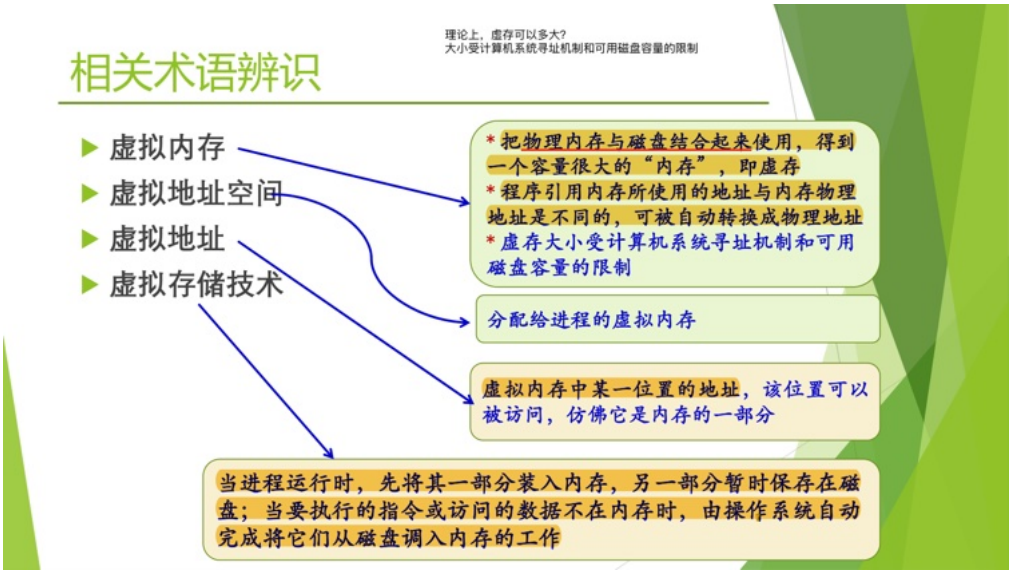
- 现代计算机存储是金字塔型：寄存器 -> L1/L2/L3 cache -> 内存 -> 本地磁盘 -> 远程存储；
- 操作系统的目标是让上层看到一个**速度接近 CPU 取指 / 容量足够大**的“内存”；
- 虚拟存储是这一抽象的核心实现。

6.4.2 概念辨析

- **虚拟存储技术**：进程运行时只把一部分装入内存，其它保存在磁盘；按需调入。
- **虚拟内存**：把物理内存与磁盘结合起来形成的“大内存”。
- **虚拟地址空间**：分配给进程的虚拟内存范围（连续编号的地址）。
- **虚拟地址**：虚拟内存中某一位置的地址。
- 虚存大小受**计算机寻址机制**和**可用磁盘容量**限制——不是无限大。



图示：寄存器 -> Cache -> 内存 -> 磁盘形成的多级存储，由 OS 协调使用，目标是“速度快 + 容量大”。



图示：虚拟内存、虚拟地址空间、虚拟地址、虚拟存储技术之间的关系。

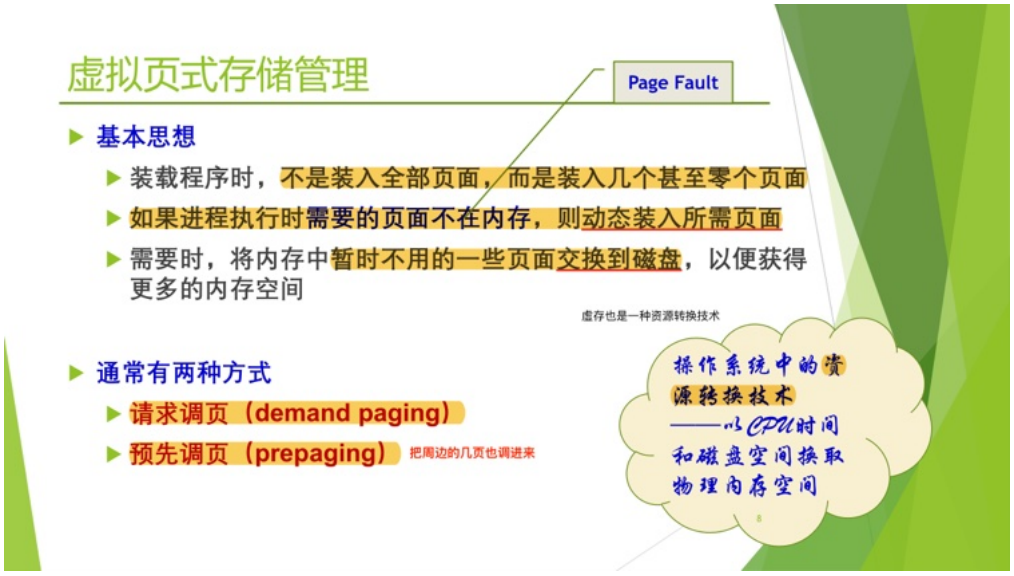
6.4.3 虚拟页式存储管理基本思想

- 装载程序时不装入全部页面（甚至零页）；
- 进程执行需要的页不在内存时，OS 自动从磁盘调入；
- 内存不足时把暂时不用的页换到磁盘。

两种调入方式：

- **请求调页（demand paging）**：访问时才调入；
- **预先调页（prepaging）**：把周边几页一起调入，利用局部性预取。

> **要点：**这是“**资源转换技术**”——用 CPU 时间（处理缺页异常）+ 磁盘空间换取物理内存空间。



图示：装入时不放入全部页，仅按需调入；不够用时换出旧页；类似 demand paging 与 prepaging。

6.4.4 调页机制：三个核心策略



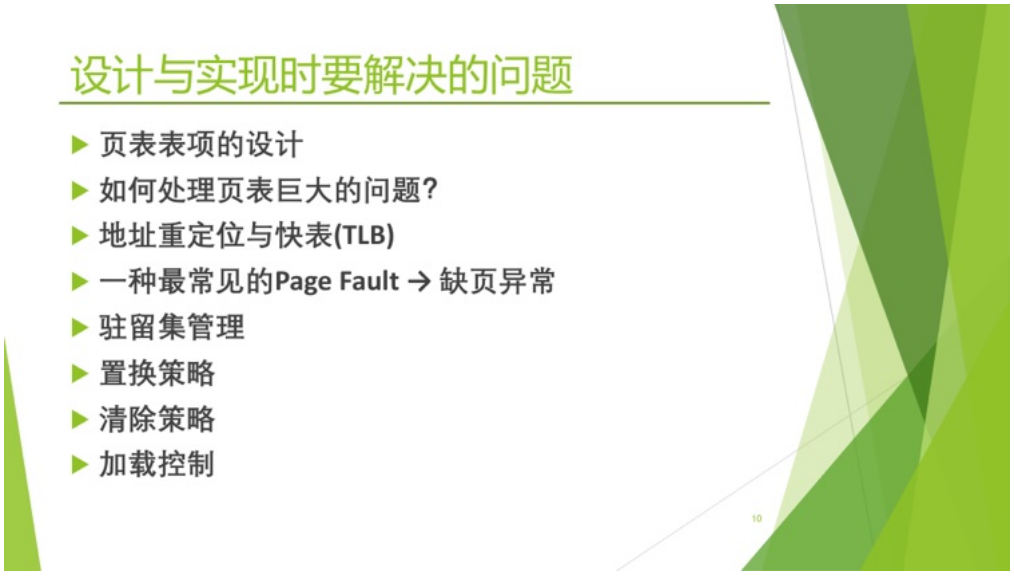
图示：取页策略、放置策略、置换策略——Coffman & Denning 提出。

调页系统包含三个核心策略：

1. **取页策略（fetch policy）**：何时把页面装入内存；
2. **放置策略（placement policy）**：把页面放在哪里；
3. **置换策略（replacement policy）**：内存不够时把哪一页“删除”。

6.4.5 设计与实现要解决的问题

后续章节将依次回答这些问题。



图示：页表项设计、页表巨大问题、地址重定位与 TLB、缺页处理、驻留集管理、置换 / 清除策略、加载控制。

6.5 页表表项设计



图示：页框号、有效位、访问位、修改位、保护位等。

PTE 通常包含：

- **页框号 PFN（Page Frame Number）**：物理页号；
- **有效位（Present, P）**：表示该页是在内存还是磁盘；
- **访问位（Accessed / Referenced, A）**：最近是否被访问过；
- **修改位（Dirty / Modified, D）**：内容是否被改过；
- **保护位**：读 / 写 / 执行 / 用户 vs 内核；
- 缓存控制位、PWT / PCD 等其他属性。

6.5.1 i386 PDE / PTE



图示：x86 PDE 和 PTE 的位格式：PFN、Avail、PS、G、A、D、PWT、PCD、R/W、U/S、P。

记忆：

- **P**：是否在物理内存中；
- **A**：最近是否被访问；
- **D**：是否被修改；
- **R/W**：可读 vs 可写；
- **U/S**：用户态 vs 内核态；
- **PS**：是否使用大页（4MB / 2MB）；
- **PWT / PCD**：缓存写回策略与禁用缓存。

6.6 多级页表

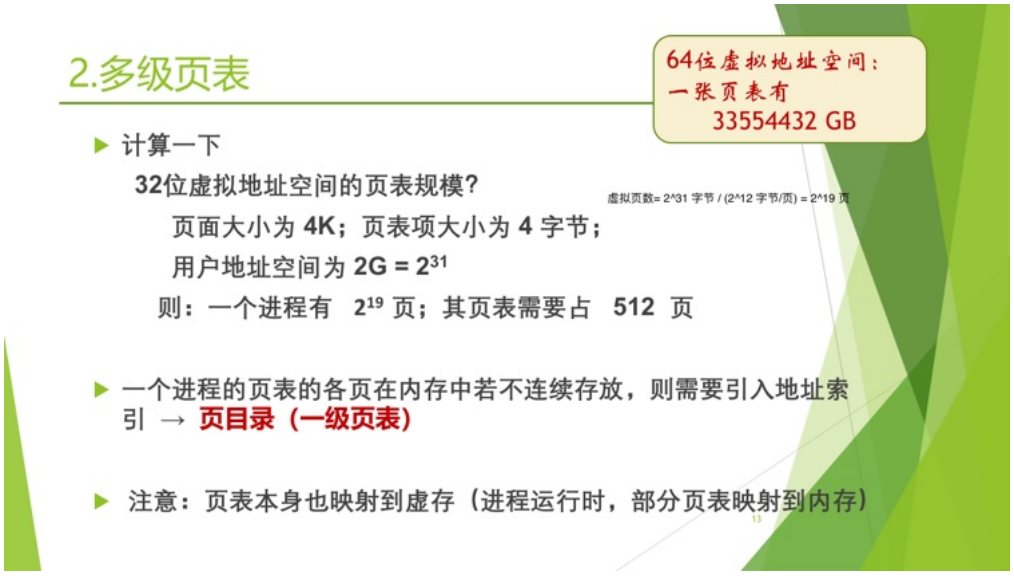
6.6.1 单级页表的规模问题

- 32 位 / 4KB 页 / 4 字节 PTE / 2GB 用户空间 -> 2^{19} 个页表项 -> 占 512 页（每页 4KB）；
- 在内存中找 512 个连续页存放页表是不现实的；
- 64 位空间下完整页表更不可行（如 33,554,432 GB）。

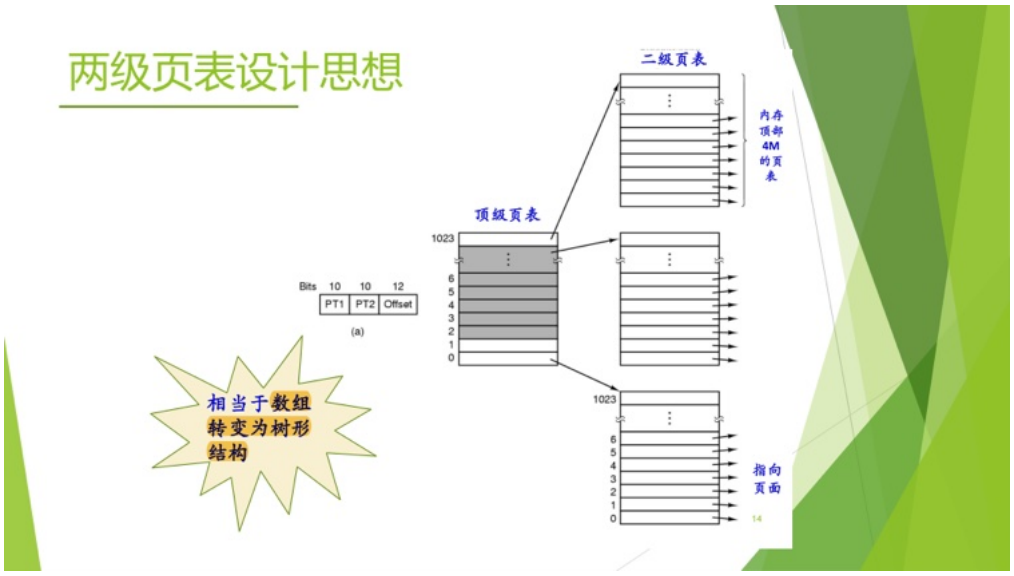
解决方案：**多级页表**——把页表本身切成页，按需驻留。

6.6.2 两级页表设计思想

- 把单层页表的“数组”改成两层“树”；
- **页目录（PDE 数组）**指向若干页表（PTE 数组）；
- 每个页表本身是一页大小，可以单独驻留 / 换出。



图示：32 位 / 4KB 页 / 4 字节 PTE / 2GB 用户空间，仍要 2^{1} 页表项；64 位下完整页表更不可行。



图示：把数组型页表转为树型，顶部的“页目录”指向若干页表。

6.6.3 两级页表的地址映射

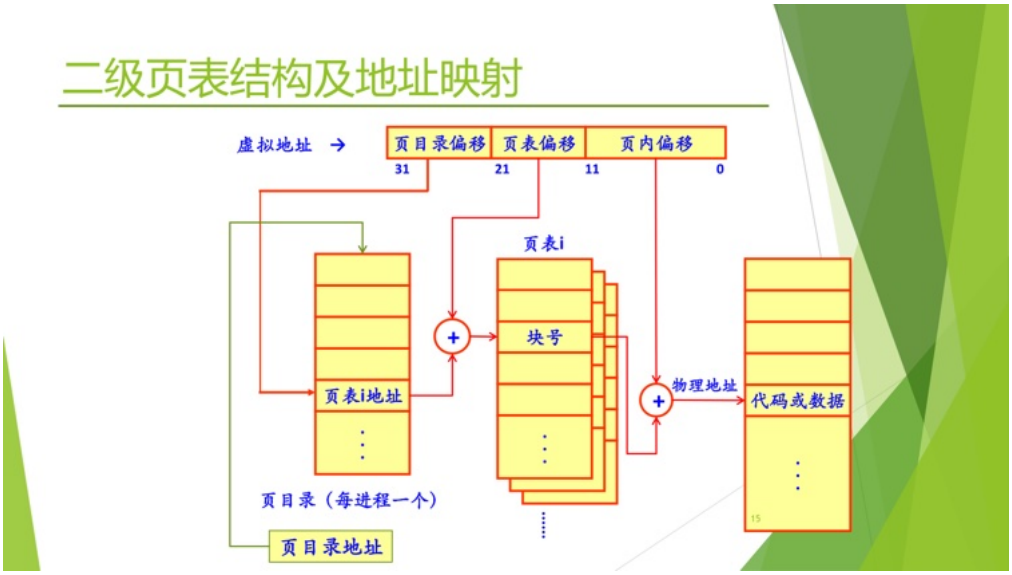
虚地址按位划分：

```
[ 页目录偏移 | 页表偏移 | 页内偏移 ]
```

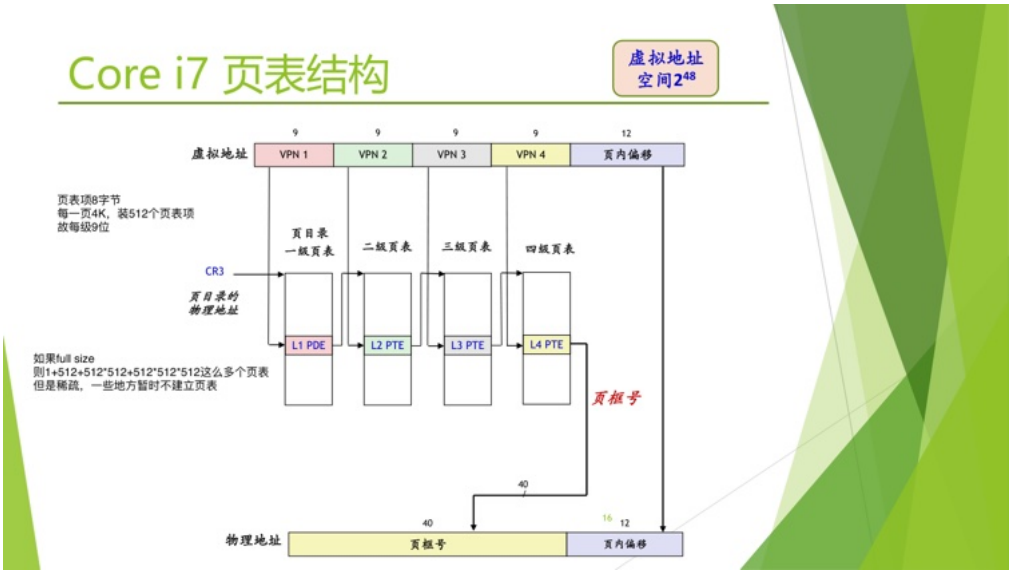
- 用页目录偏移在页目录中找到对应的 PTE 数组（页表）；
- 用页表偏移在页表中找到 PFN；
- PFN + 页内偏移 = 物理地址。

6.6.4 Core i7 四级页表

- x86-64 / Core i7 使用四级页表；



图示：虚地址 = 页目录偏移 + 页表偏移 + 页内偏移；通过两次查表得到物理页框号。



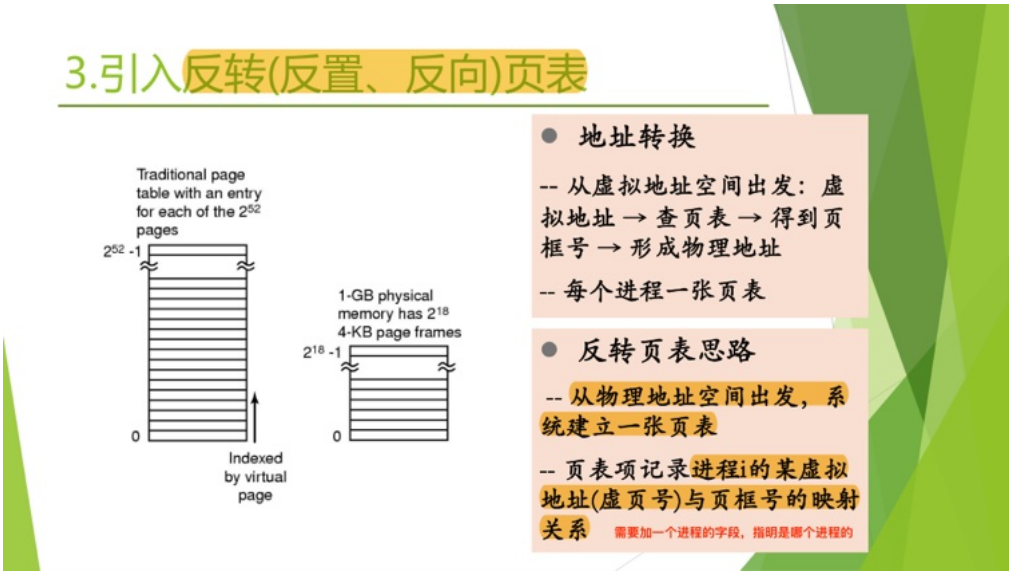
图示：CR3 -> 页目录 -> 三级 -> 二级 -> 一级 -> 4KB 页；每级 9 位，共 36 位虚拟页号 + 12 位页内偏移。

- 每级 9 位，PTE 8 字节，每页可装 512 项；
- 四级总共映射 2 字节虚拟空间；
- 页表稀疏：实际只为有用的虚拟区域建立页表，绝大部分虚拟空间不分配 PTE。

> **小算：**如果四级全建满，需要 $1 + 512 + 512^2 + 512^3$ 张页表；但稀疏存储后实际只需要常数级数量。

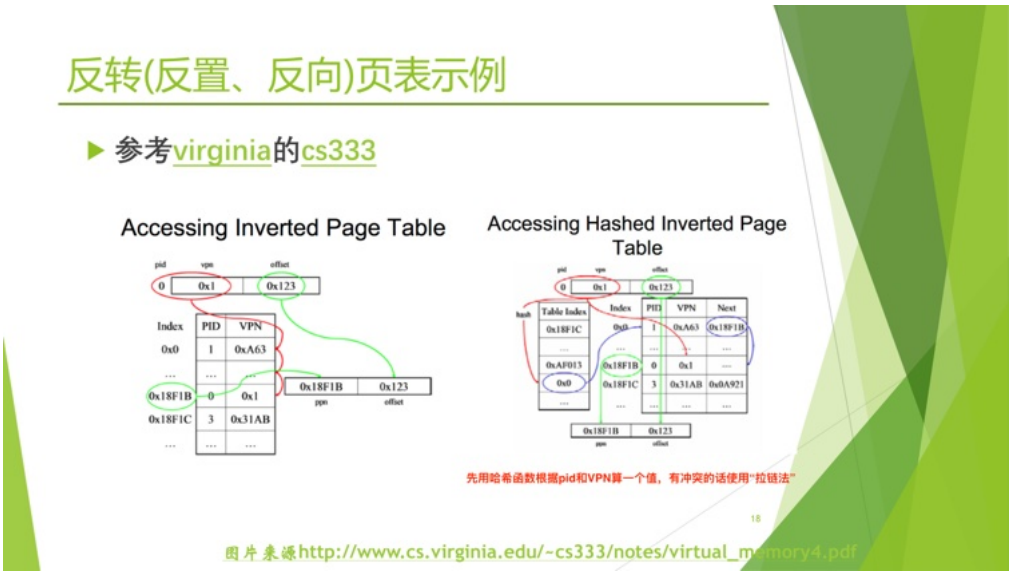
6.7 反转 / 反置页表

- 传统页表：每个进程一张表，从虚拟地址出发查找物理页；
- **反置页表（Inverted Page Table）**：以物理页为索引，记录“物理页 -> 进程 + 虚拟页”；
- 因此页表项中需要加一个进程标识（如 PID）。



图示：传统页表从虚拟地址出发，每进程一张；反置页表从物理地址出发，全系统一张。

6.7.1 实现机制

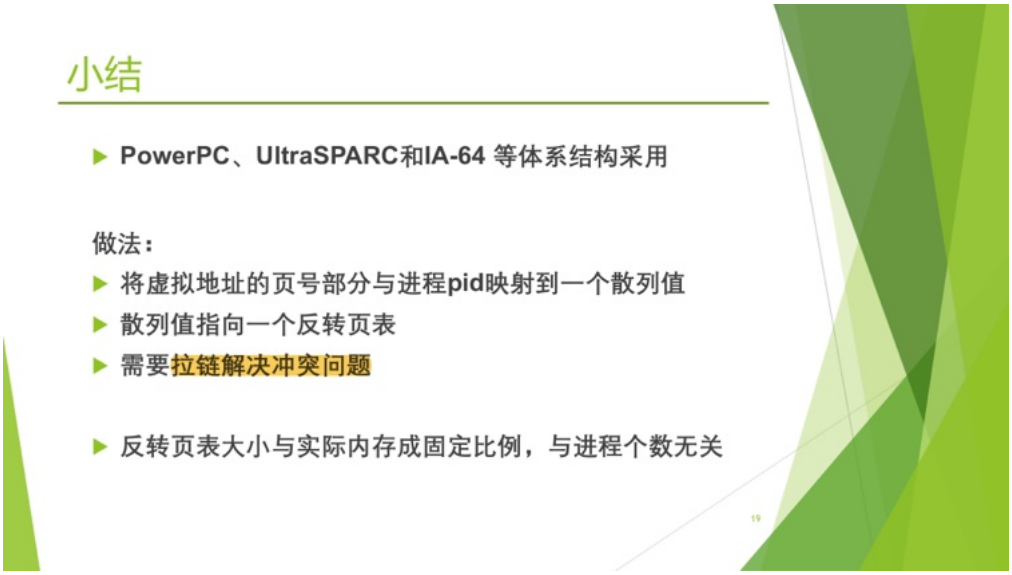


图示：通过哈希函数把（PID, VPN）映射到反置页表的桶；冲突使用拉链法。

- 用 hash(PID, VPN) 得到反置页表的索引；
- 同桶冲突用链表链接；
- 反置页表的大小与物理内存成比例，与进程数无关；
- 适合大内存、多进程的服务器场景。

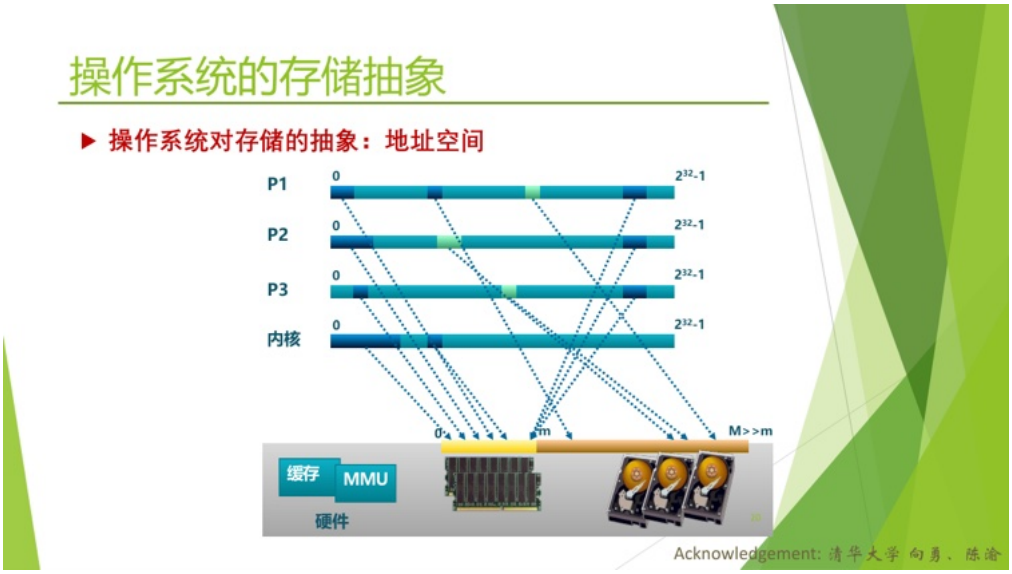
6.7.2 应用

- PowerPC、UltraSPARC、IA-64 等使用反置页表；
- 优点：节省页表存储；
- 缺点：每次访问都要 hash + 拉链查找，硬件 / 软件实现复杂。



图示：PowerPC、UltraSPARC、IA-64 等架构使用反置页表。

6.8 操作系统的存储抽象与 MMU



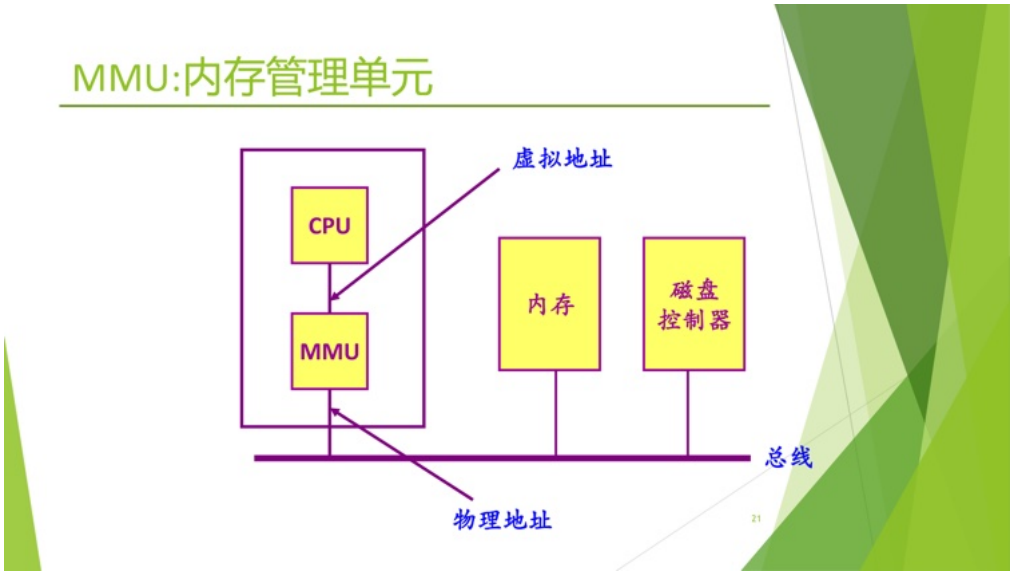
图示：每个进程看到 $0 \sim 2^{32}-1$ 的虚拟空间，由 MMU 翻译到不同物理位置；$m \ll M$。

- MMU 位于 CPU 与内存之间；
- 每条访存指令都经过 MMU；
- MMU 失败（页不在内存、权限不符）就触发缺页 / 保护异常。

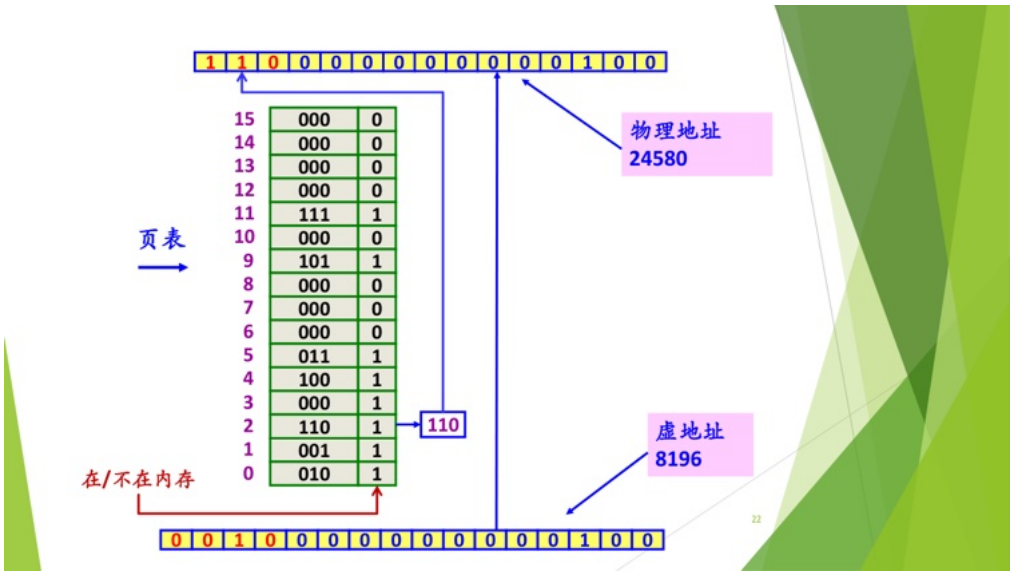
6.9 地址转换流程

伪代码：

```
if (虚拟页面不在内存 / 页面非法 / 受保护) {
    硬件触发异常，进入操作系统的 Page Fault 处理
```



图示：CPU -> MMU -> 内存 -> 磁盘控制器，MMU 负责把虚拟地址转换为物理地址。



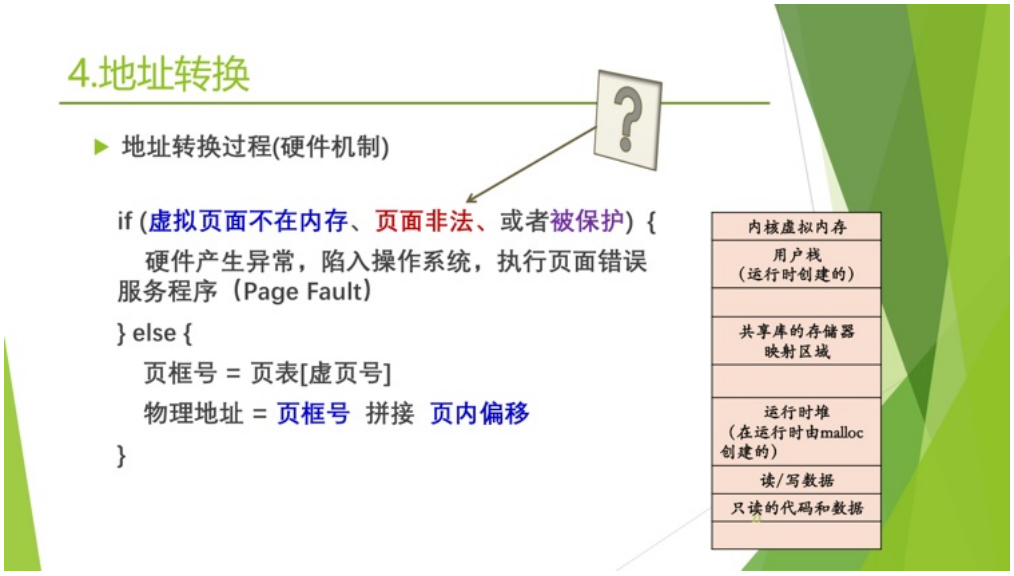
图示：每个 PTE 有“是否在内存”的位；虚地址 8196 -> 通过页表得到物理地址 24580。

```
} else {
    页框号 = 页表[虚页号]
    物理地址 = 页框号 << 页内偏移位 + 页内偏移
}
```

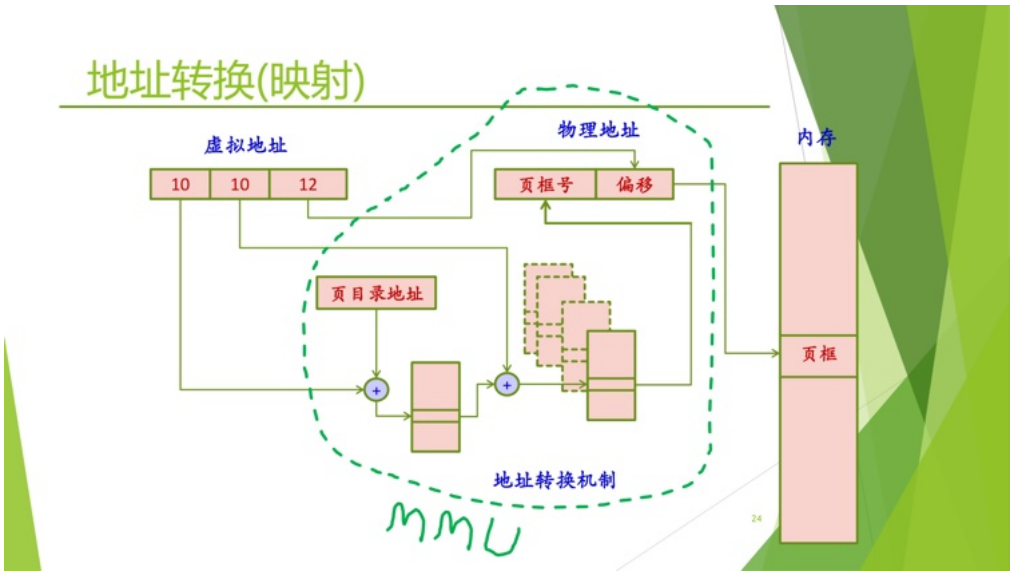
6.10 快表 TLB

6.10.1 引入

- 多级页表带来的额外内存访问让访存代价倍增；
- 程序局部性原理意味着同一组页表项会被反复使用；
- 引入 **TLB**（Translation Look-aside Buffer），存近期使用的 PTE 子集。



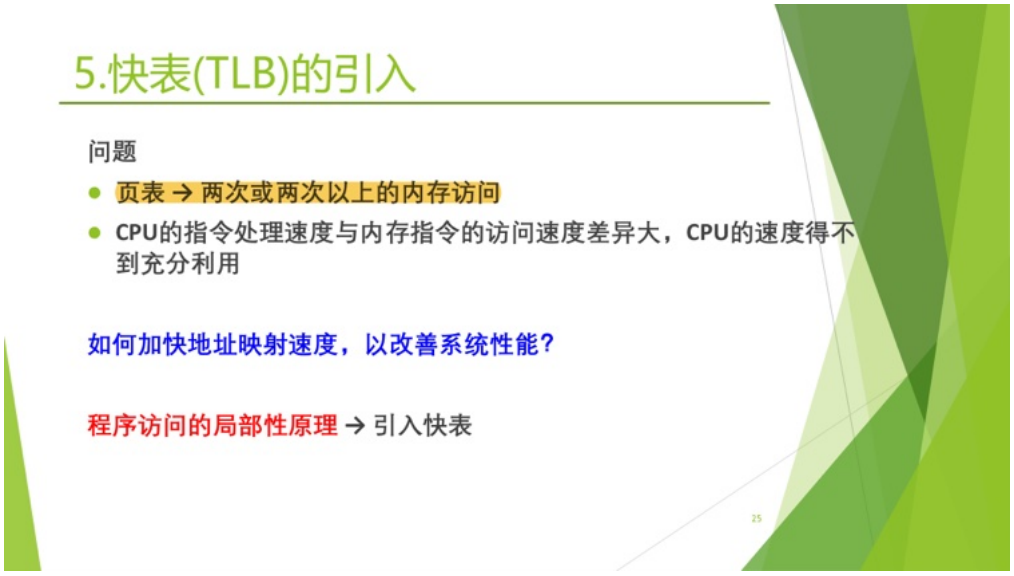
图示：if 不在内存 / 非法 / 受保护 -> 触发 Page Fault；else 拼接页框号与偏移得到物理地址。



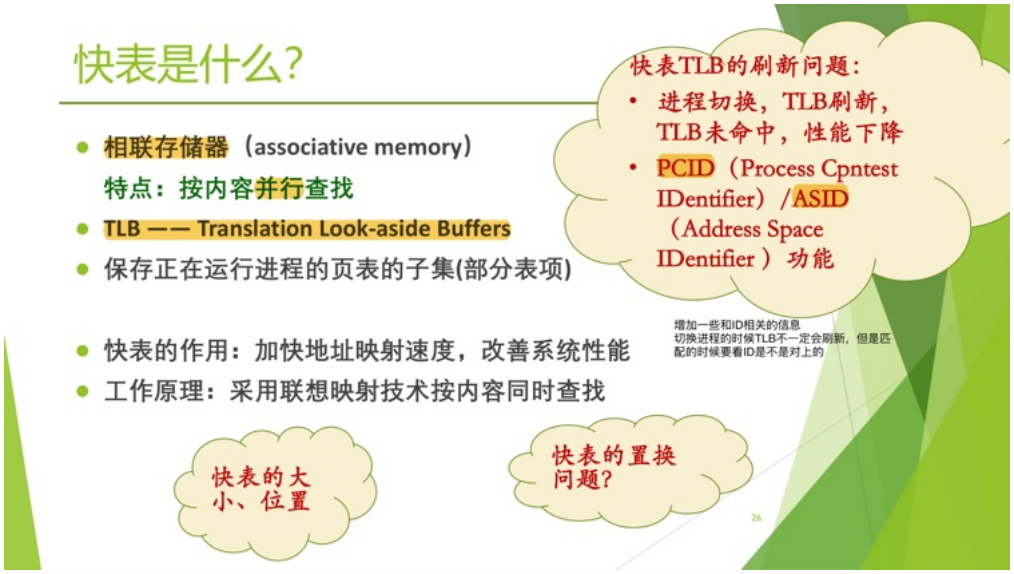
图示：32 位下，10 + 10 + 12 划分为页目录偏移、页表偏移、页内偏移；两次查表 + 偏移拼接。

6.10.2 TLB 是什么

- 相联存储器（associative memory），按内容并行查找；
- 保存当前正在运行进程的页表子集；
- 命中时 1 个周期完成地址转换；
- 命中失败（TLB miss）时回退到查多级页表，并把结果加入 TLB；
- TLB 大小有限（如 64 / 128 / 1024 项）；
- 切换进程时通常要刷 TLB，影响性能；
- **PCID / ASID** 功能：给每个 TLB 项加进程 ID 标记，避免每次切换都刷 TLB。



图示：每次地址转换都要 2 次或更多内存访问；引入快表利用局部性原理加速。



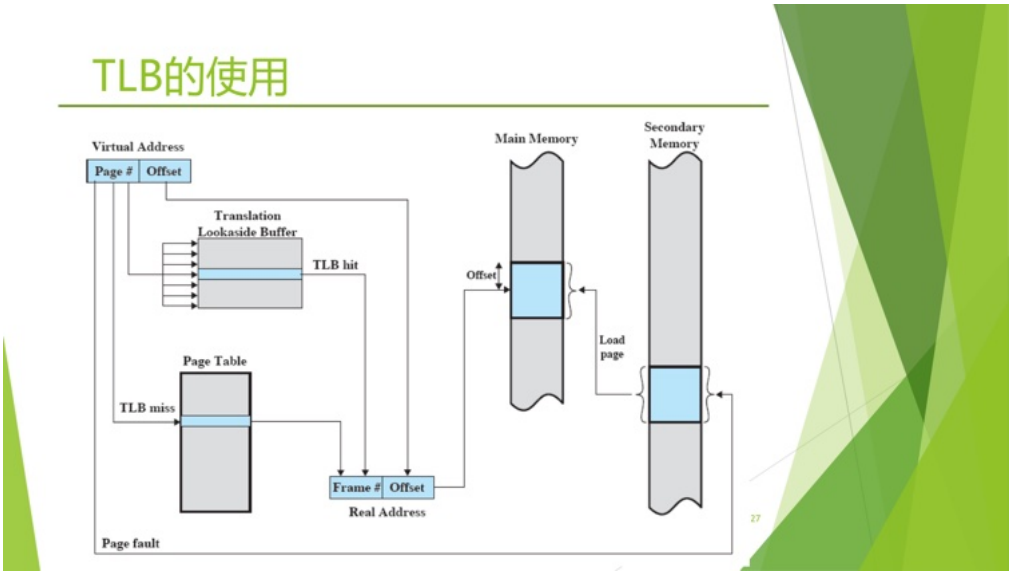
图示：相联存储器，按内容并行查找；保存当前进程的页表子集。

6.10.3 TLB 的使用与高速缓存的关系

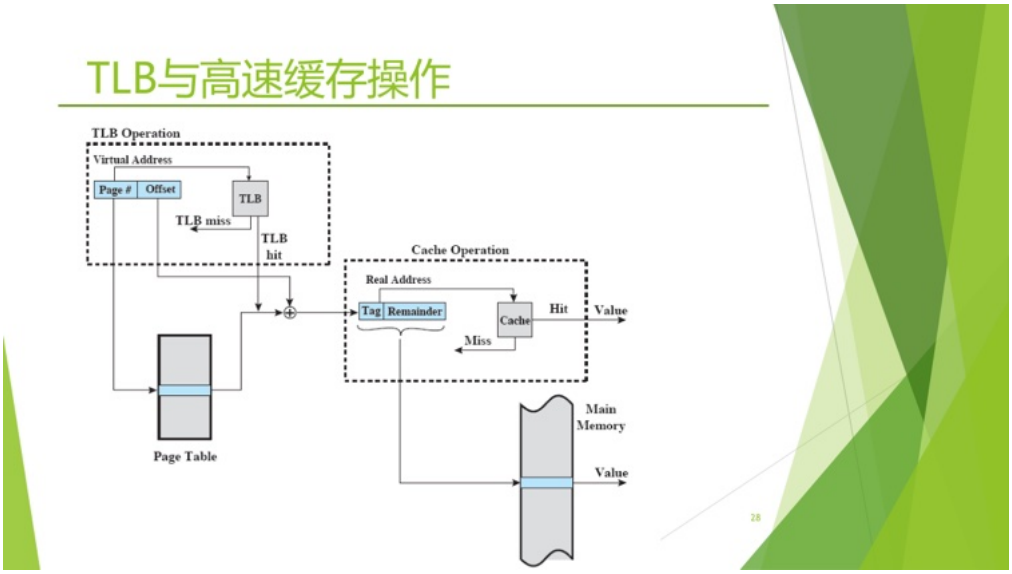
6.11 缺页异常处理

缺页异常处理流程：

1. 地址转换中硬件发现 PTE 无效（不在内存或权限不符）；
2. 触发硬件异常，进入 OS 缺页处理程序；
3. OS 解析异常原因，定位磁盘地址，启动磁盘把页调入内存；
4. 若内存有空闲页框，则分配一页，把数据装入，更新 PTE（驻留位、PFN）；
5. 若没有空闲页框，则按置换策略选一页换出；
6. 若换出页是脏页，先写回磁盘；



图示：CPU 把虚地址送到 TLB；命中直接得到物理地址；未命中走多级页表。

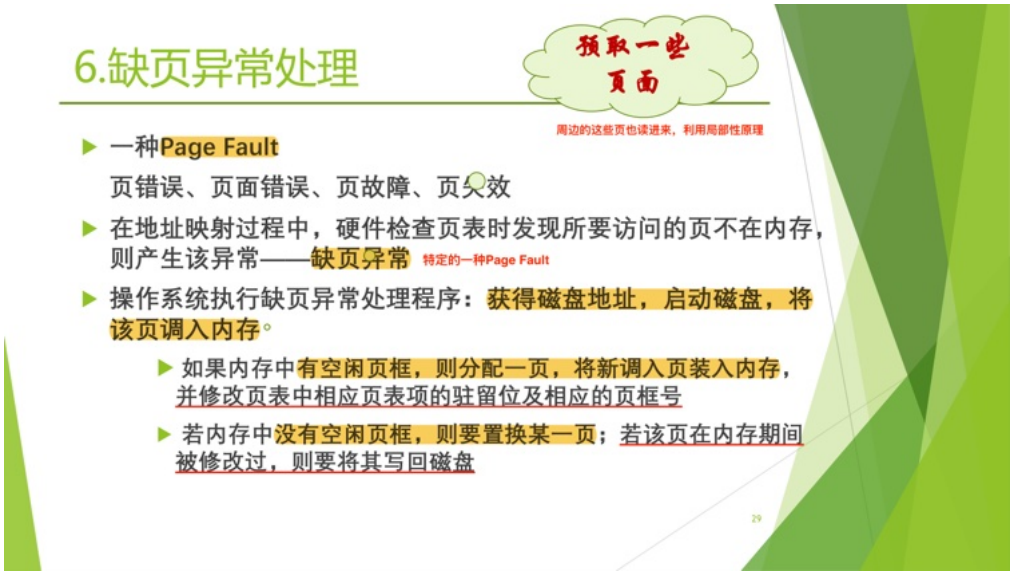


图示：在物理地址形成后再去查 L1/L2/L3 cache；TLB 是地址翻译的 cache。

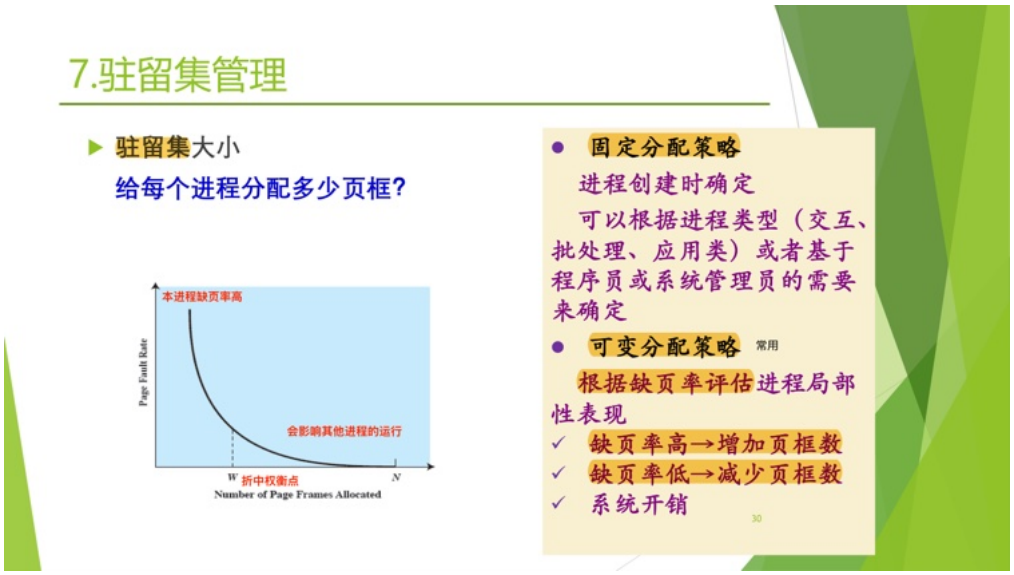
7. 调入完成后，OS 让被异常打断的指令重试；
8. 可顺便预取邻近若干页，利用局部性。

6.12 驻留集管理

- **驻留集（Resident Set）**：当前驻留在物理内存中的页框集合。
- 大小决定策略：
  - **固定分配**：进程创建时确定驻留集大小，不再改变；
  - **可变分配**：根据缺页率动态评估：
    - 缺页率高 -> 增加页框；
    - 缺页率低 -> 减少页框；



图示：硬件检测页不在内存 -> 异常 -> OS 缺页处理 -> 找空闲页框或置换 -> 调入磁盘 -> 重试访问。



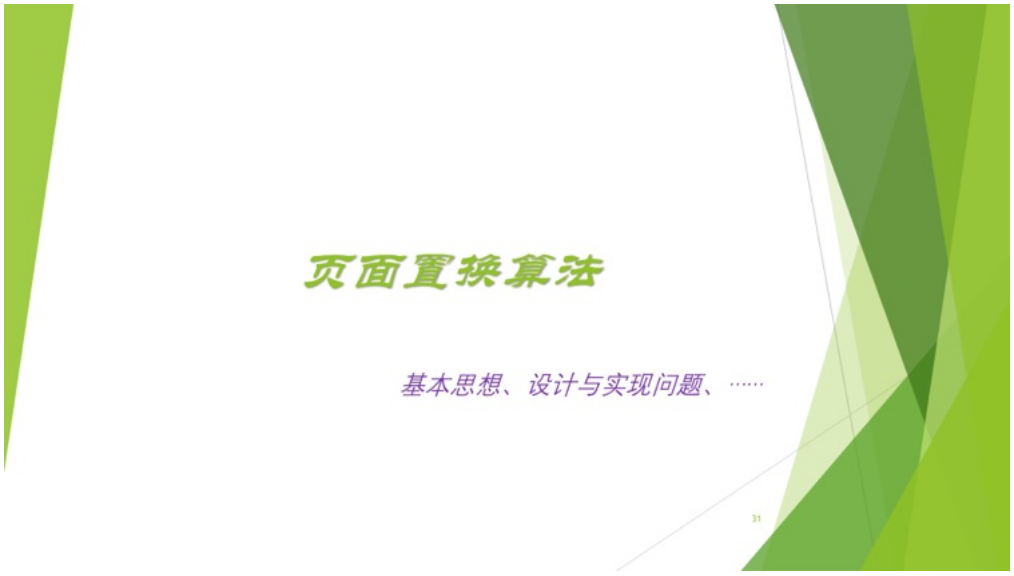
图示：固定分配 vs 可变分配；可变分配按缺页率评估。

* 但调整本身有系统开销，需要折中。

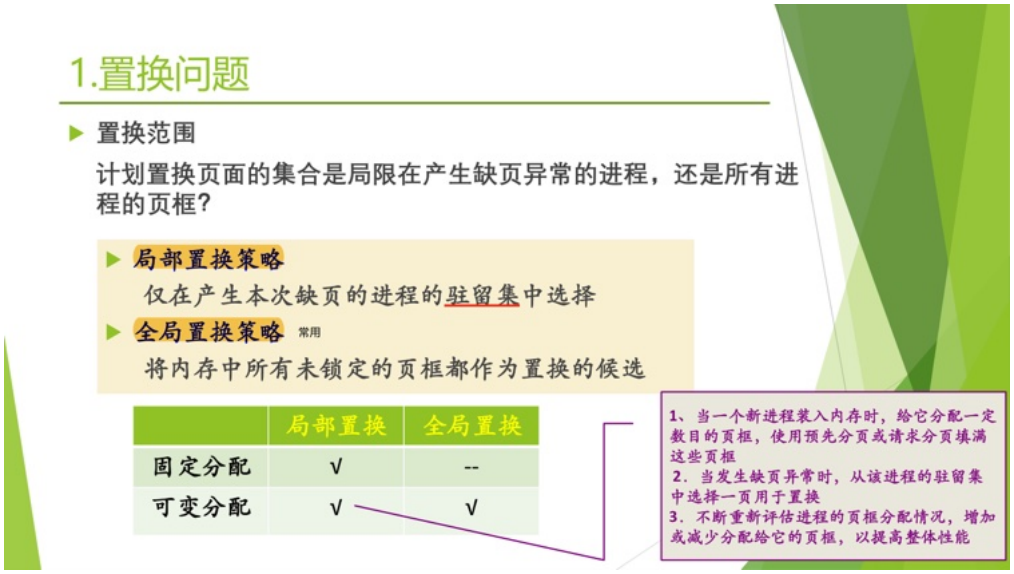
6.13 页面置换算法

6.13.1 置换问题

- **局部置换**：仅在当前缺页进程的驻留集中选；
- **全局置换**：把内存中所有未锁定页框都作为候选；
- **置换 + 分配组合**：



图示：置换问题、各种典型算法。

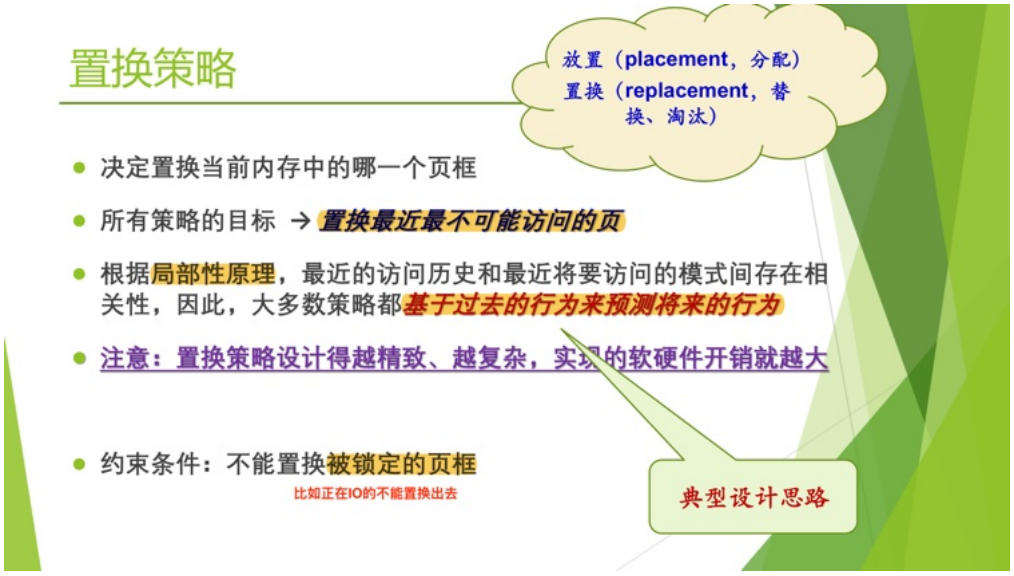


图示：局部置换 vs 全局置换；与固定 / 可变分配的组合表。

分配策略	局部置换	全局置换
固定分配	√	—
可变分配	√	√（常用）

- 流程：装入新进程时按预先 / 请求分页填满页框；缺页时从驻留集选一页换出；不断重新评估页框分配。

- 所有策略的目标：置换最近最不可能访问的页；
- 基于**局部性原理**，过去与将来访问模式相关；
- 设计越精致，开销越大；
- **不能置换被锁定的页框**（如正在 I/O 的页）。



图示：置换“最近最不可能访问的页”；策略越精致，软硬件开销越大。

6.13.2 页框锁定



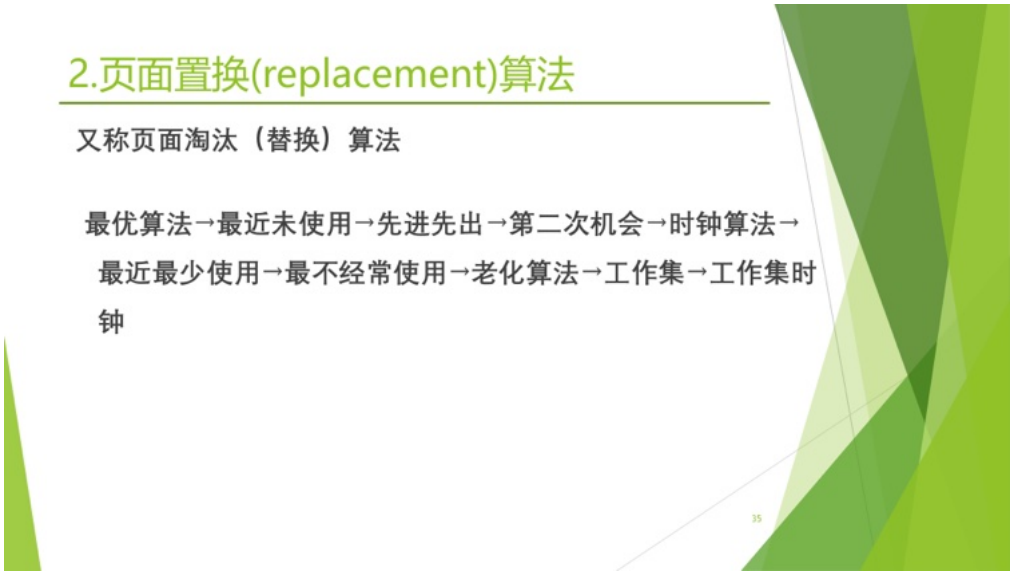
图示：每个页框增加锁定位；锁定 OS 核心代码、关键数据结构、I/O 缓冲区等。

- 给每个页框加“锁定位”；
- 锁定的页面不会被换出；
- 典型场景：操作系统核心代码、关键数据结构、I/O 缓冲区；
- Windows 提供 `VirtualLock` / `VirtualUnlock`。

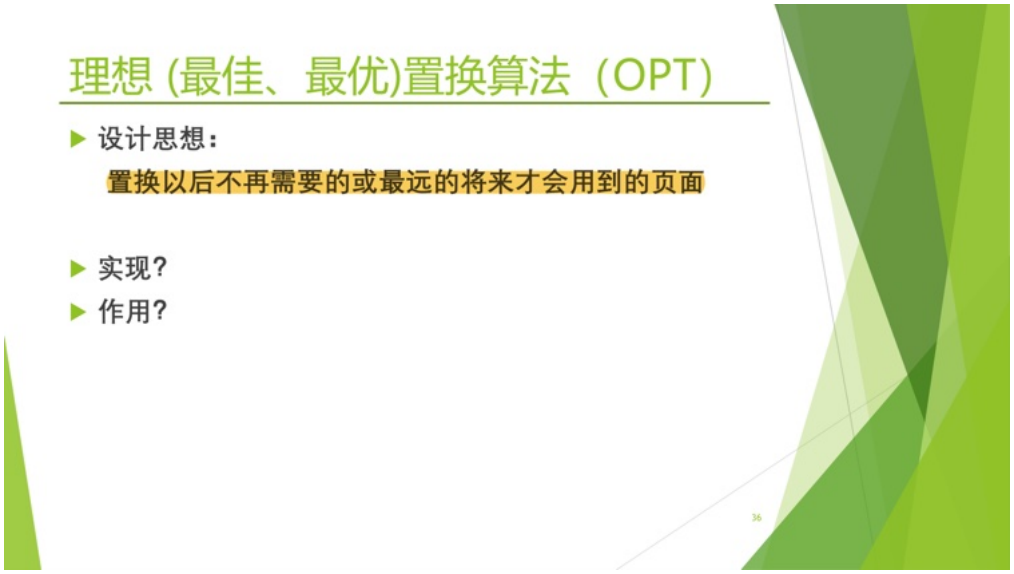
6.13.3 各种置换算法

最优算法 OPT

- 置换“以后不再需要的或最远的将来才会用到”的页面；
- **不可实现**，因为需要预知未来；



图示：从 OPT -> NRU -> FIFO -> 第二次机会 -> 时钟 -> LRU -> NFU -> 老化 -> 工作集 -> 工作集时钟。



图示：选未来最长时间内不会被访问的页置换；不可实现，但可作为基准。

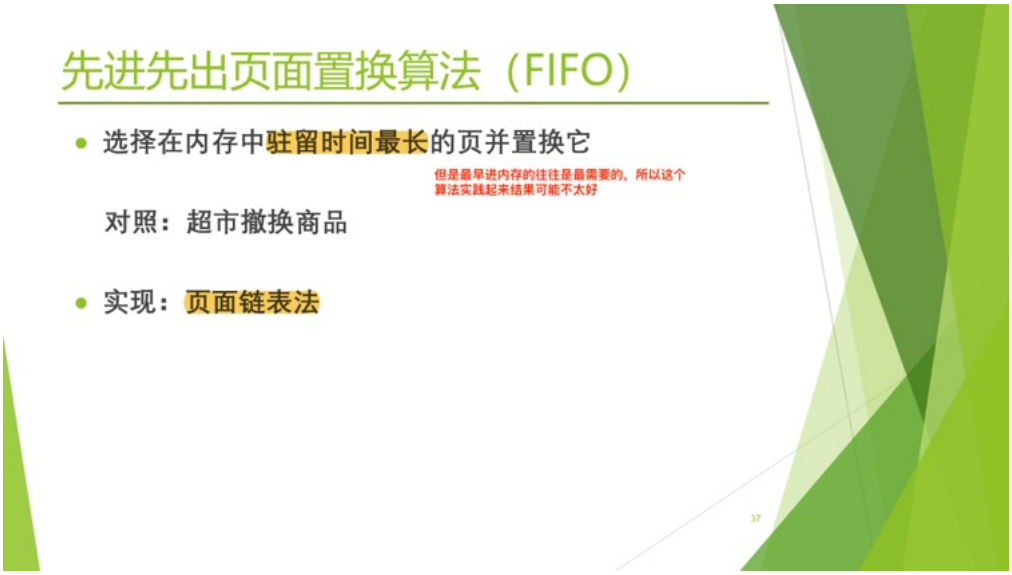
- 用作衡量其他算法的理论基准。

先进先出 FIFO

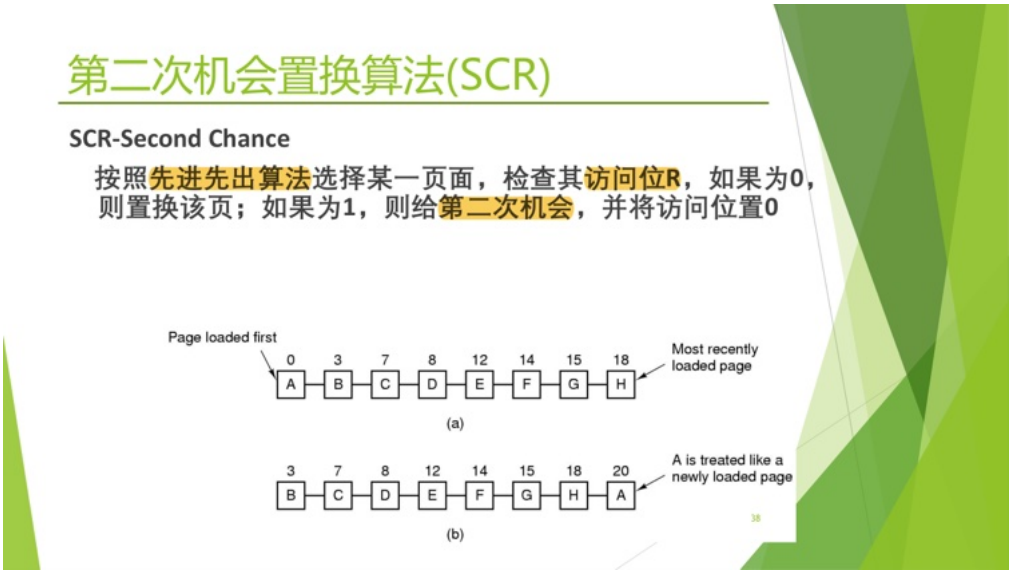
- 选驻留时间最长的页置换；
- 实现：页面链表；
- 缺点：最早进入的页面常常仍在被使用，置换效果不好；可能出现 Belady 异常。

第二次机会 SCR

- 在 FIFO 选中的页面上检查访问位 R；
- 如果 R=0，置换；
- 如果 R=1，给“第二次机会”，把 R 置 0，让它继续等。



图示：选驻留时间最长的页置换；用页面链表实现。



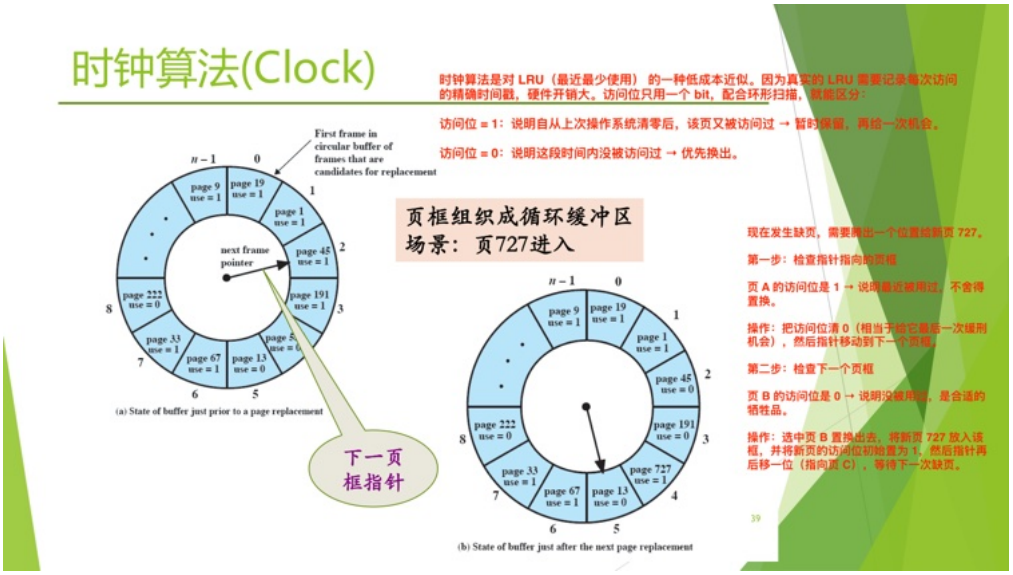
图示：FIFO + 检查访问位 R；R=1 给“第二次机会”，并把 R 置 0。

时钟算法 Clock

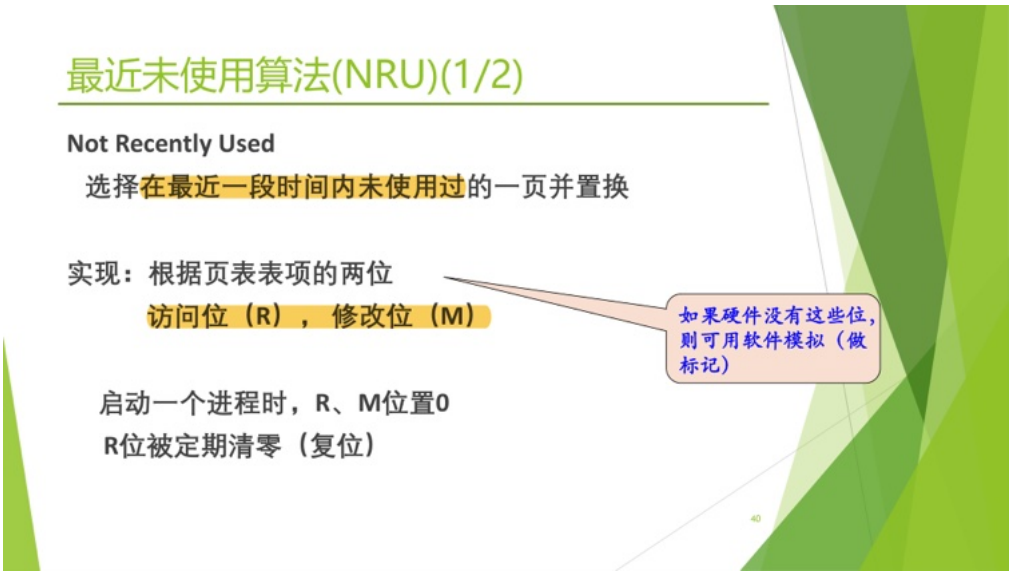
- 时钟算法是 LRU 的低成本近似；
- 真实 LRU 需要每次访问都记时间戳，硬件代价大；
- 访问位仅用 1 bit + 环形扫描就能近似：
  - R=1：最近被访问 -> 清零，跳过；
  - R=0：未被访问 -> 选作牺牲；
- 例子：727 进入，循环缓冲区中找到 R=0 的页框替换。

最近未使用 NRU

- 每页两位：访问位 R、修改位 M；
- 四类：



图示：把页面组织成循环缓冲区，指针扫描；R=1 清零并跳过；R=0 选作牺牲品。

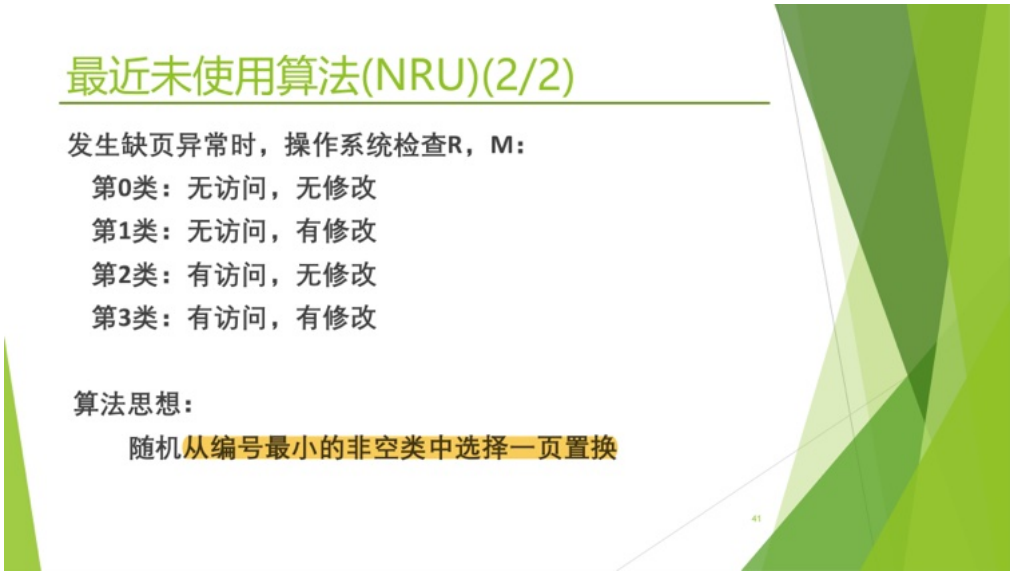


图示：基于 R + M 两位将页分四类，从编号最小的非空类中选。

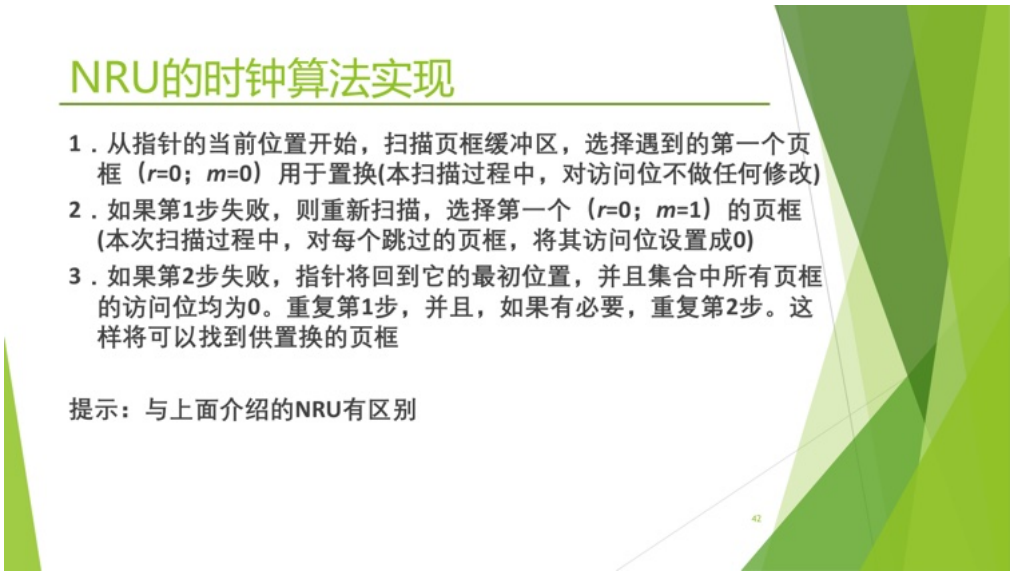
  - 第 0 类：无访问、无修改；
  - 第 1 类：无访问、有修改；
  - 第 2 类：有访问、无修改；
  - 第 3 类：有访问、有修改；
- 算法：随机从编号最小的非空类中选一页置换。

- 启动进程时 R、M 均置 0；
- R 位定期清零（OS 周期任务）；
- 硬件没有这两位时可用软件模拟。

- 第一轮：扫描，选 (R=0, M=0)；这一轮不修改 R；
- 第二轮：选 (R=0, M=1)；这一轮把跳过的 R 置 0；



图示：清零策略与具体实现。



图示：分两轮扫描，分别查找 (R=0, M=0) 与 (R=0, M=1) 的页。

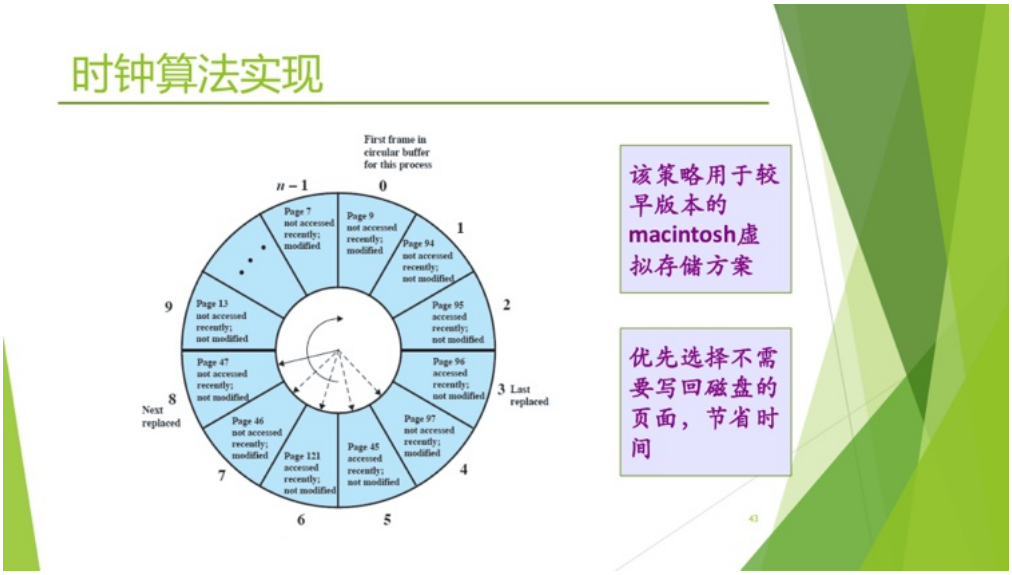
- 如仍找不到，回到第一轮再来一遍——此时所有 R 已清 0。

LRU

- LRU 性能接近 OPT；
- 实现两条思路：
  - 给每页加访问时间戳，每次访问更新；
  - 维护按访问顺序的栈；
- 都需要每次访问都更新数据，硬件支持成本高。

例子：访问顺序 0,1,2,3,2,1,0,3,2,3。

- 用 n×n 矩阵硬件维护访问关系，每次访问 i 时把第 i 行置 1、第 i 列置 0；



图示：早期 macintosh 虚存方案使用，优先选择不需要写回的页节省时间。



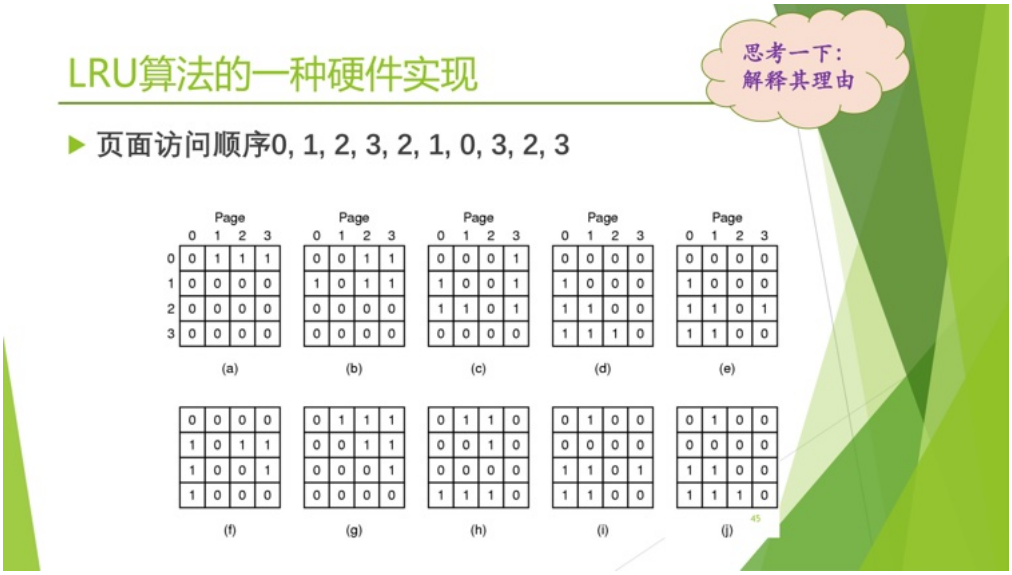
图示：选最久未访问的页置换；性能接近 OPT；实现要时间戳或访问栈，开销大。

- 矩阵中行最小的就是最久未用的页；
- 思路是用空间换实现简单。

NFU 与老化算法

- NFU（Not Frequently Used）：每页一个计数器，每次时钟中断把 R 累加到计数器；
- 选计数器最小者置换；
- 缺点：早期被频繁访问的页计数器很大，即便后面不再被用也不容易换出。

- 改进：每次时钟中断先把计数器右移 1 位，再把 R 放到最高位；
- 这样新访问权重最大，老访问权重指数衰减；
- 是 LRU 的低成本近似，常用于教学操作系统。



图示：n×n 矩阵法 + 访问顺序示例。



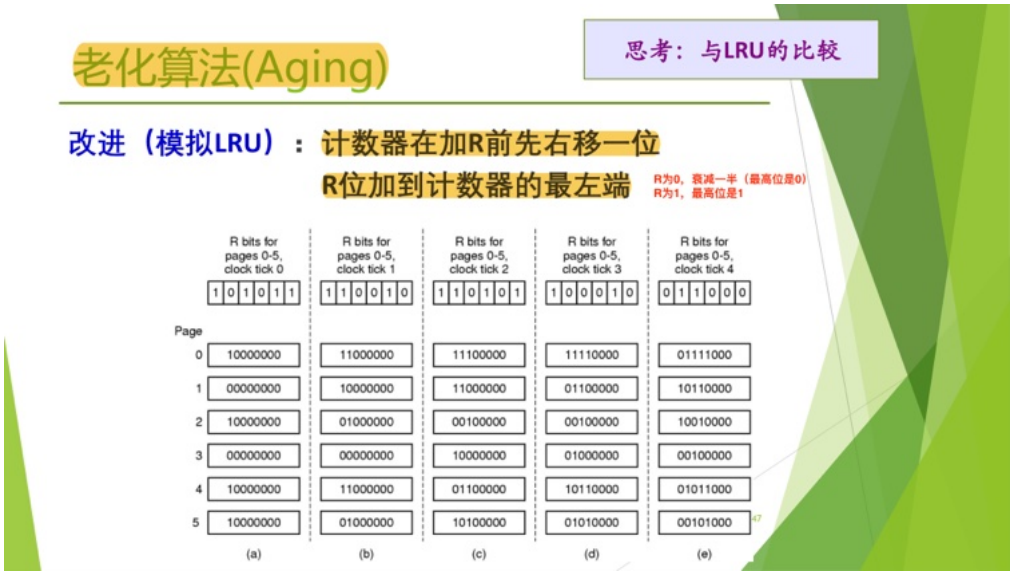
图示：每页一个软件计数器，时钟中断时累加 R；选最小者置换。

应用举例

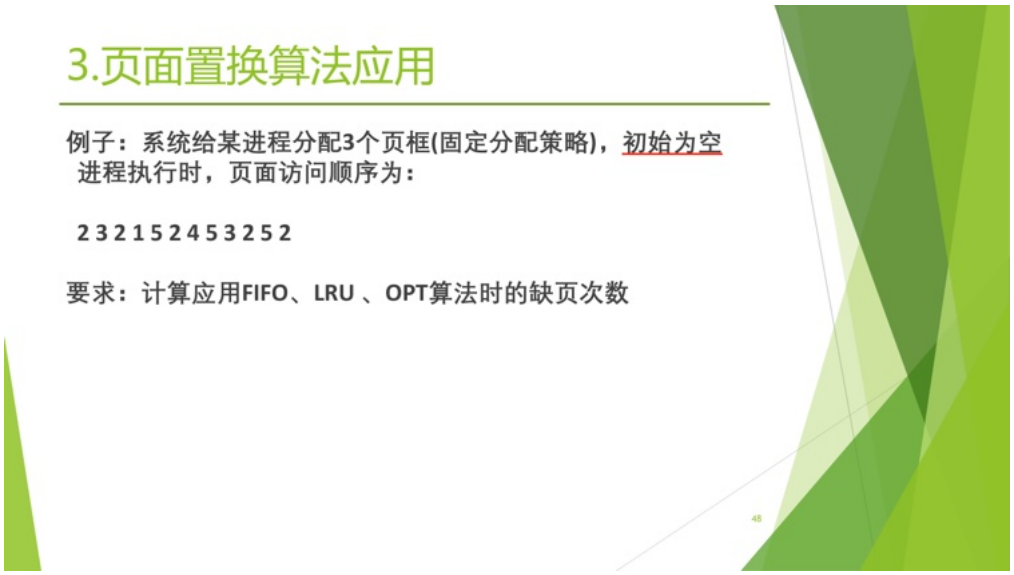
> **要点：**例题练熟后，答题就变成机械手工——每一步先看是否命中、未命中则按算法选牺牲页、画状态表。

Belady 现象

- 经典反例：序列 1 2 3 4 1 2 5 1 2 3 4 5；
- m=3 时缺页 9 次；m=4 时缺页 10 次；
- 即“**资源更多反而更差**”，是 FIFO 类算法不满足栈算法性质的体现；
- LRU、OPT 不会出现 Belady 现象。



图示：在加 R 之前先右移；R 加到最高位；近期访问权重比早期访问高。

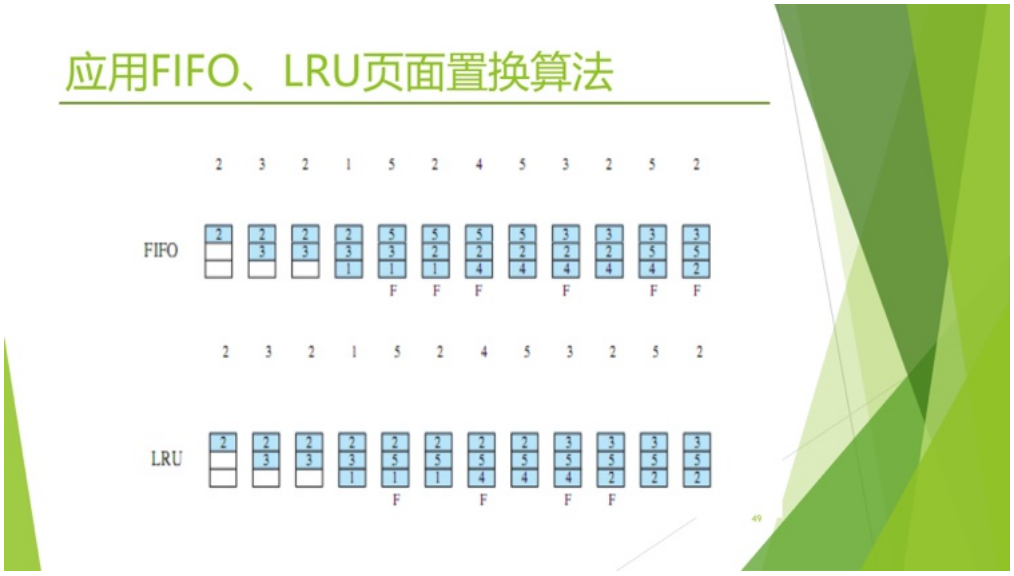


图示：3 个页框，访问顺序 2 3 2 1 5 2 4 5 3 2 5 2，分别用 FIFO/LRU/OPT 计算缺页次数。

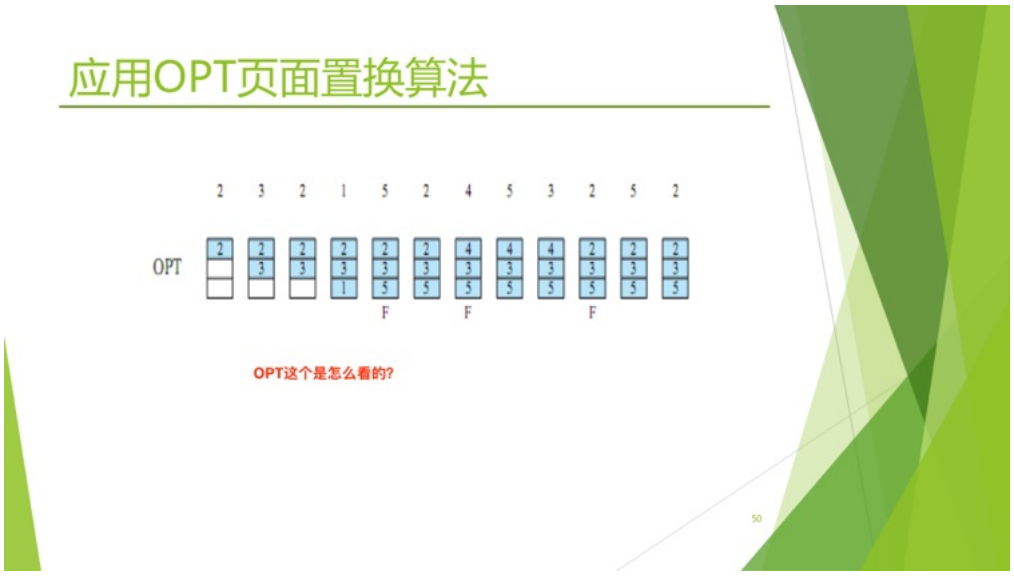
6.13.4 影响缺页次数的因素

- 页面置换算法；
- 页面大小（影响内碎片、TLB 命中率）；
- 程序编制方法（行优先 vs 列优先访问）；
- 分配给进程的物理页框数；
- **颠簸 / 抖动（Thrashing）**：页面在内存与磁盘之间频繁调度，调度时间多于实际运行时间，效率急剧下降。

- 小页：内碎片少、页表长；
- 大页：页表短、TLB 命中高，但内碎片大；
- Intel 80x86 / Pentium 同时支持 4KB 与 4MB；



图示：按列推演缺页次数。



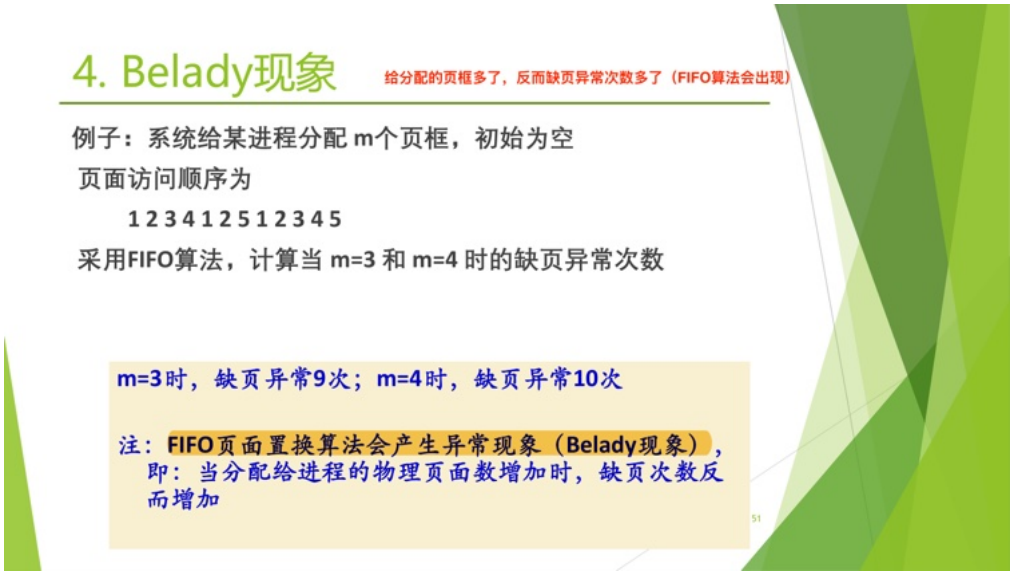
图示：按列推演 OPT 在每次缺页时如何选最远将访问的页。

- 多种页面尺寸：TLB 利用率提高，但 OS 实现复杂。

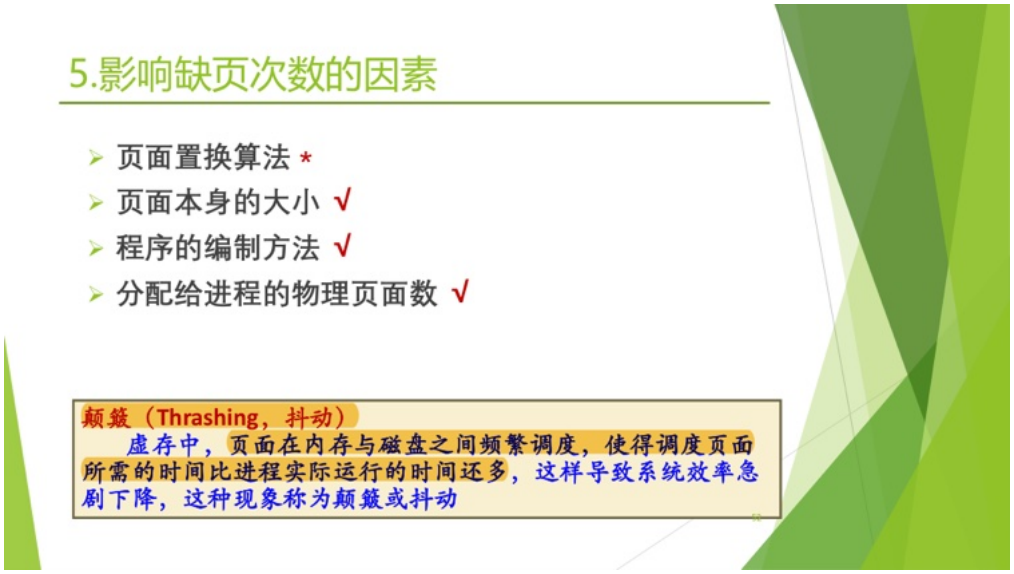
- 程序员决定数据访问顺序：
  - 方法 1：先列后行 -> 几乎每次都缺页；
  - 方法 2：先行后列 -> 利用局部性大幅减少缺页；
- 这就是“**为什么矩阵乘法要按 cache 友好顺序写**”的根本原因。

6.14 工作集模型

- **工作集（Working Set）**由 Denning 1968 提出；
- 思想：进程在某段时间集中访问一组页面（活跃页面）；
- 给进程分配“活跃页面 <= 页框数”就能让缺页率显著下降。



图示：FIFO 下，分给进程的页框增加，缺页次数反而增加。



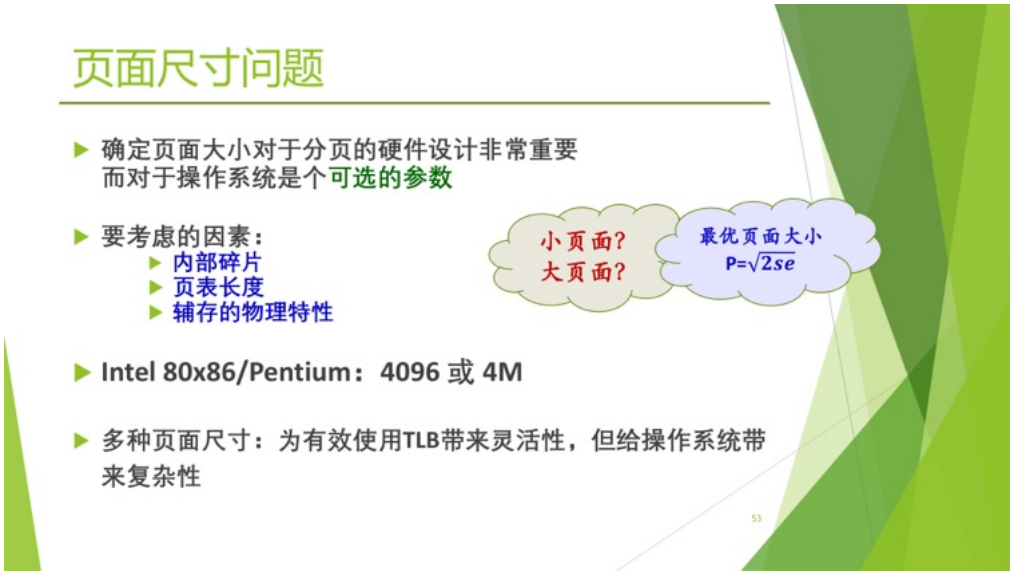
图示：置换算法、页面大小、程序编制方法、分配的物理页数；以及抖动现象。

工作集 $W(t, \Delta)$ 取决于三个因素：

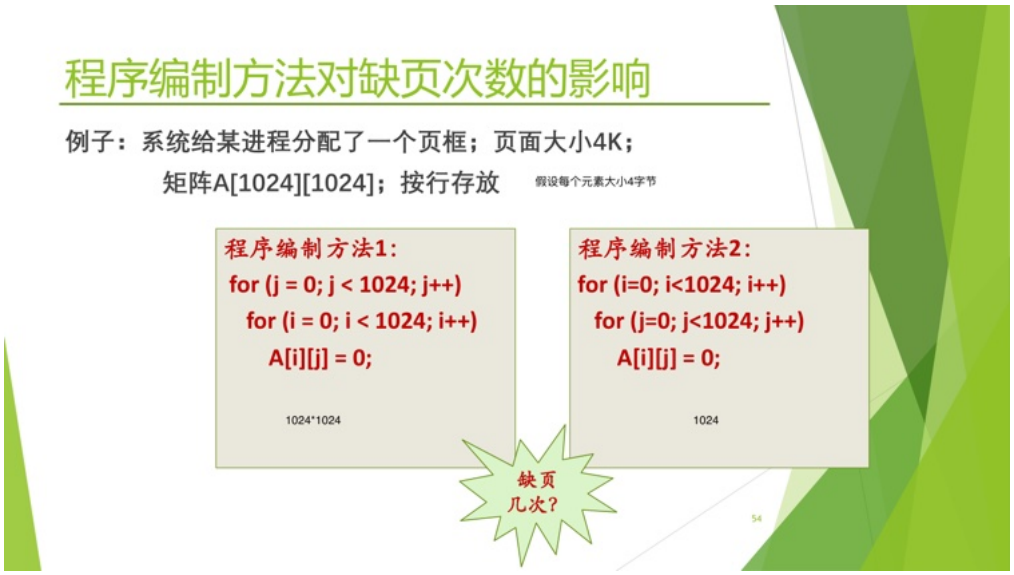
- 访页序列特性；
- 当前时刻 t；
- 工作集窗口长度 Δ；
- Δ 越大，工作集越大；过大易包含很多无用页，过小易漏掉常用页。

例子：访问序列 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 7，窗口 Δ=10：

- $W(t1, 10) = \{1, 2, 5, 6, 7\}$
- $W(t2, 10) = \{3, 4\}$



图示：内碎片 vs 页表长度 vs 辅存物理特性的权衡，以及最优页面大小公式。



图示：1024×1024 矩阵 A，按行还是按列访问，缺页次数相差 1024 倍。

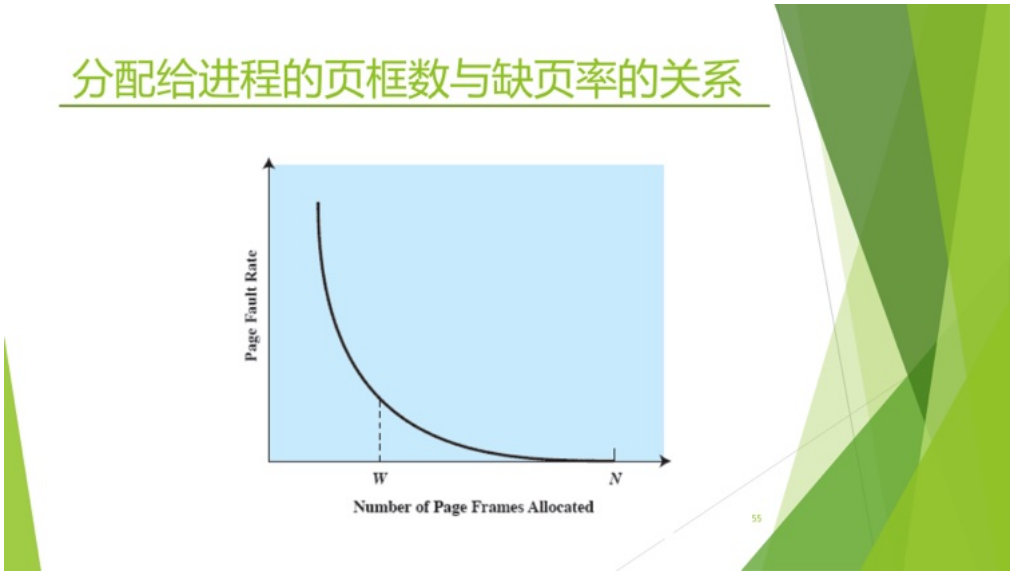
6.14.1 工作集与驻留集

- **工作集**：进程运行的内在属性；
- **驻留集**：实际分给进程的页框集合；
- 监视每个进程的工作集，可用 LRU 周期性把不在工作集中的页移出驻留集；
- 当进程的工作集都在内存时，缺页率最低。

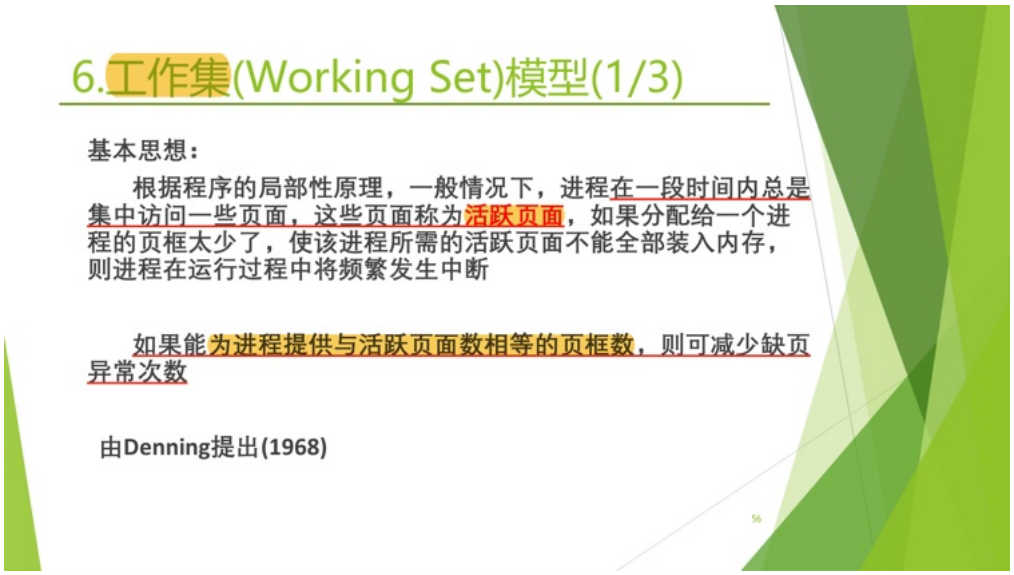
6.14.2 工作集算法

实现细节：

1. 扫描每个页表项；
2. 若 R=1：把最后访问时间设为当前时间，把 R 清零（“最近来过的房间”擦掉对勾）；



图示：缺页率随页框数增加而下降，但有边际效应，超过某个值后收益迅速变小。

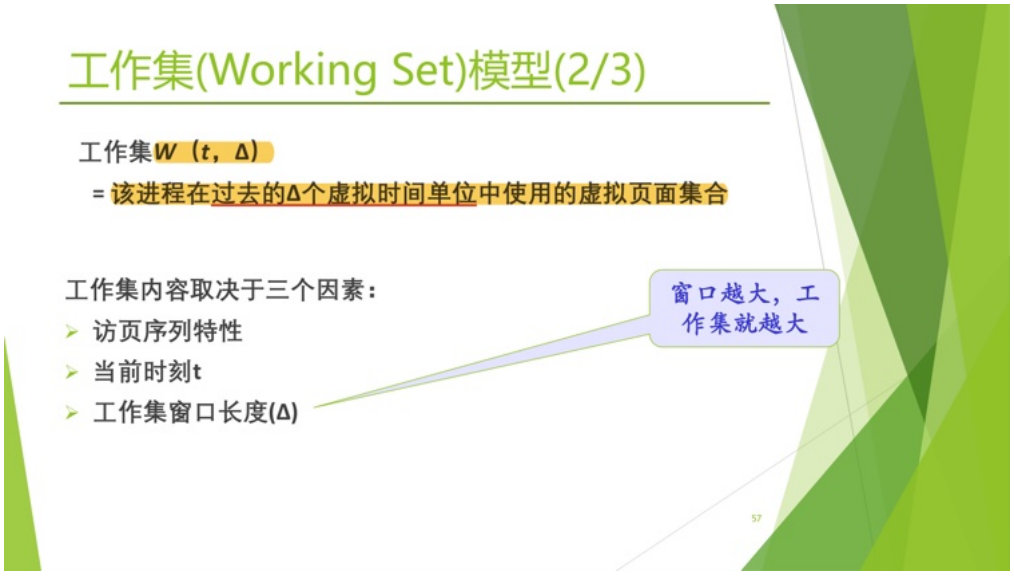


图示：根据局部性，进程一段时间集中访问一些“活跃页面”；分配活跃页面数量的页框可减少缺页。

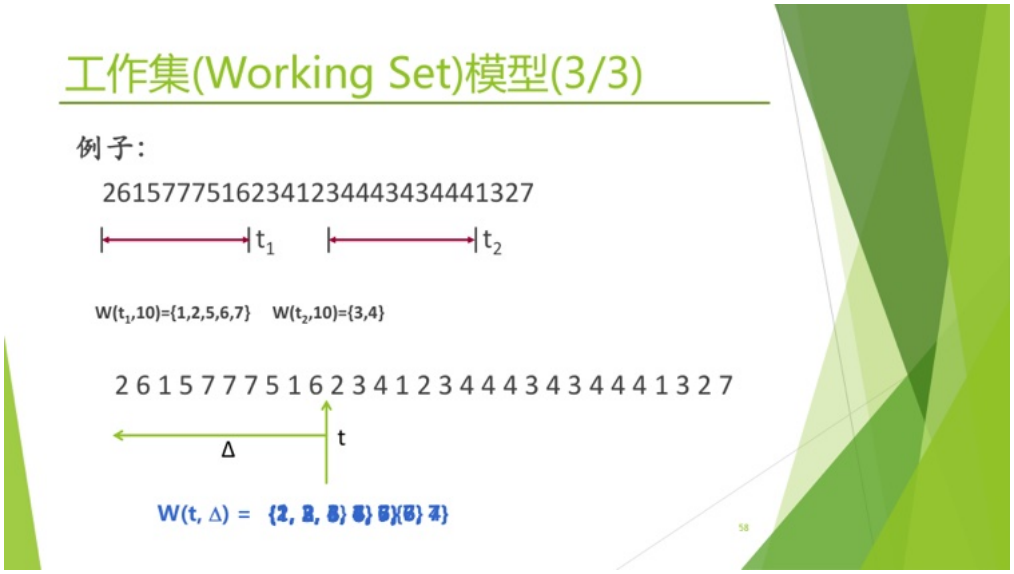
3. 若 R=0：检查最后访问时间是否在“当前时间 − T”之前：
   - 是 -> 选作置换页；
   - 否 -> 记录最小最后访问时间，扫描下一页。
4. 找不到可置换页，就选最小访问时间的页置换。

6.14.3 缺页率算法

- 当驻留集变大时，缺页率下降；
- 当驻留集小到工作集都装不下时，缺页率显著上升；
- 设阈值上下限：
  - 缺页率 > 上限 -> 增加页框；

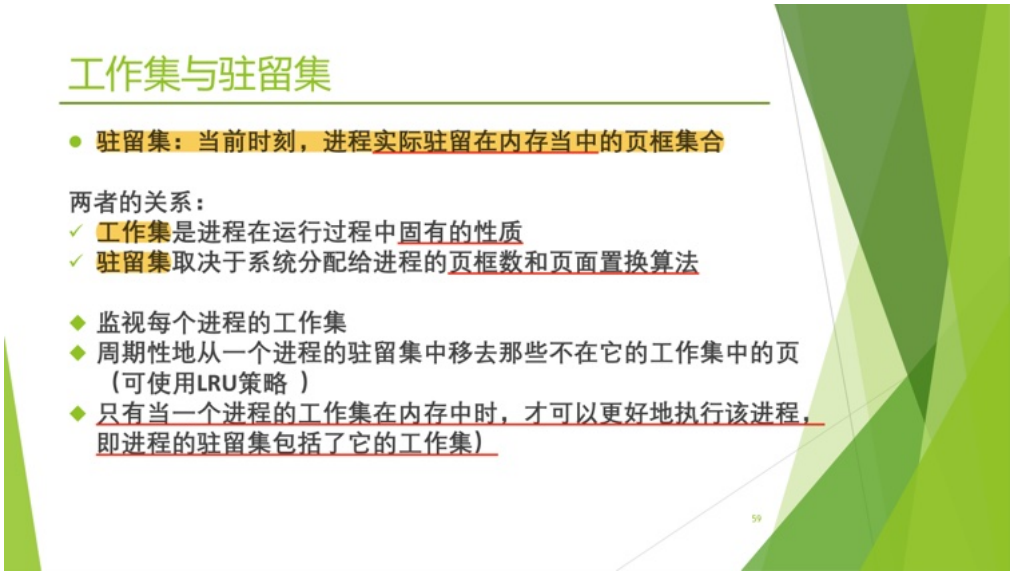


图示：W(t, Δ) 是进程过去 Δ 个虚拟时间单位内使用过的虚拟页面集合。

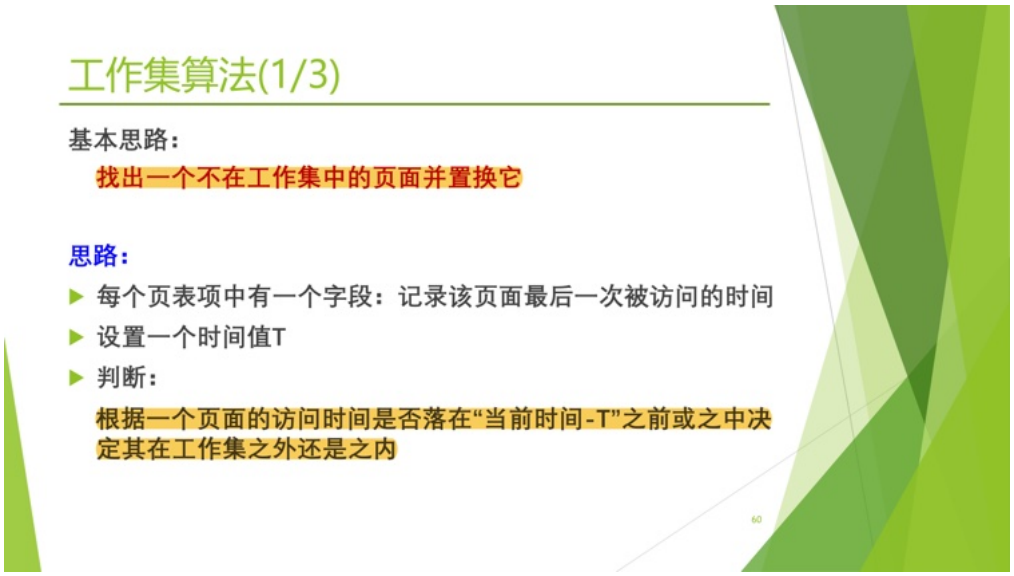


图示：访问序列示例 + 不同时刻 / 窗口下的工作集。

  - 缺页率 < 下限 -> 减少页框；
- 这是工作集思想的工程化近似，许多 OS 都采用。



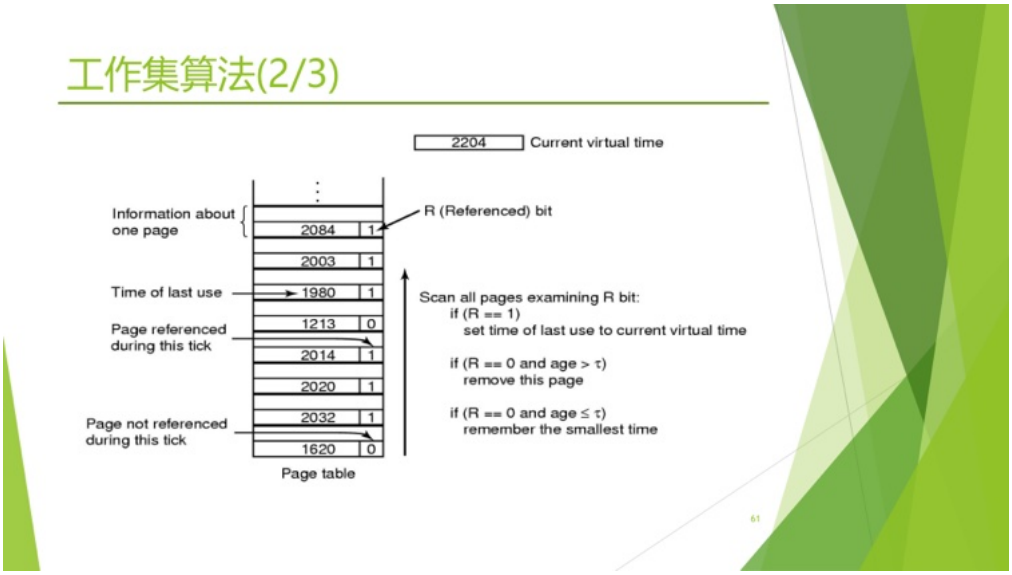
图示：工作集是进程内在性质；驻留集取决于系统页框数和置换算法。



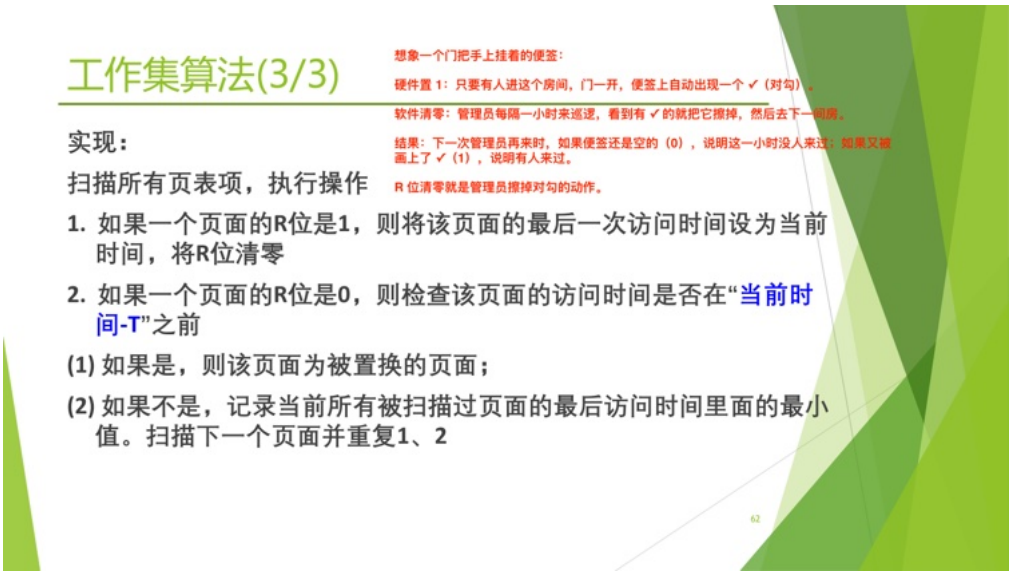
图示：在每个 PTE 中记录最后一次访问时间；用阈值 T 判断是否在工作集内。

6.14.4 算法小结

算法	评价
OPT	不可实现，但可作基准
NRU	LRU 的非常粗略近似
FIFO	可能淘汰重要页面，存在 Belady
Second Chance	比 FIFO 改进显著
Clock	现实中常用
LRU	优秀但难实现
NFU	LRU 的相对粗略近似
Aging	非常近似 LRU 的有效算法
Working Set	实现开销大
WSClock	高效且实用



图示：扫描所有页框，根据 R 位与最后访问时间判断置换候选。



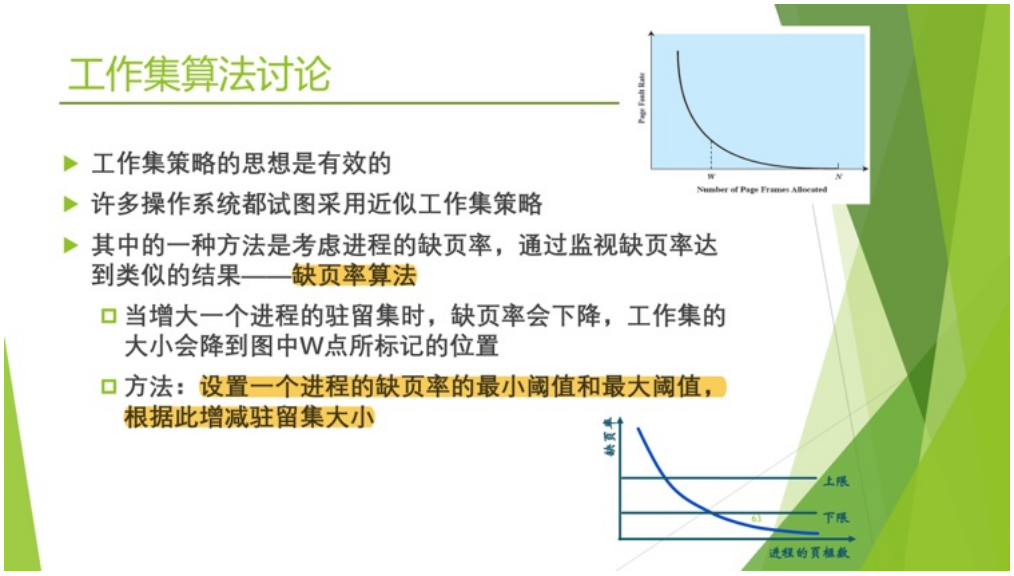
图示：硬件置 R=1，软件周期性清零，判断访问时间是否在“当前时间 − T”之前。

6.15 清除策略与加载控制

6.15.1 清除策略—分页守护进程

- 分页系统最佳状态：发生缺页时有大量空闲页框；
- 设计分页守护进程（paging daemon）周期性醒来：
  - 检查空闲页框数量；
  - 数量过少时按置换算法把脏页写回磁盘并加入空闲池；
  - 让所有空闲页框都是“干净”的。

- 双指针时钟：
  - 前指针：分页守护进程控制，遇脏页就写回，遇干净页只移动；



图示：通过监视缺页率达到类似效果；设定上下阈值动态增减驻留集。

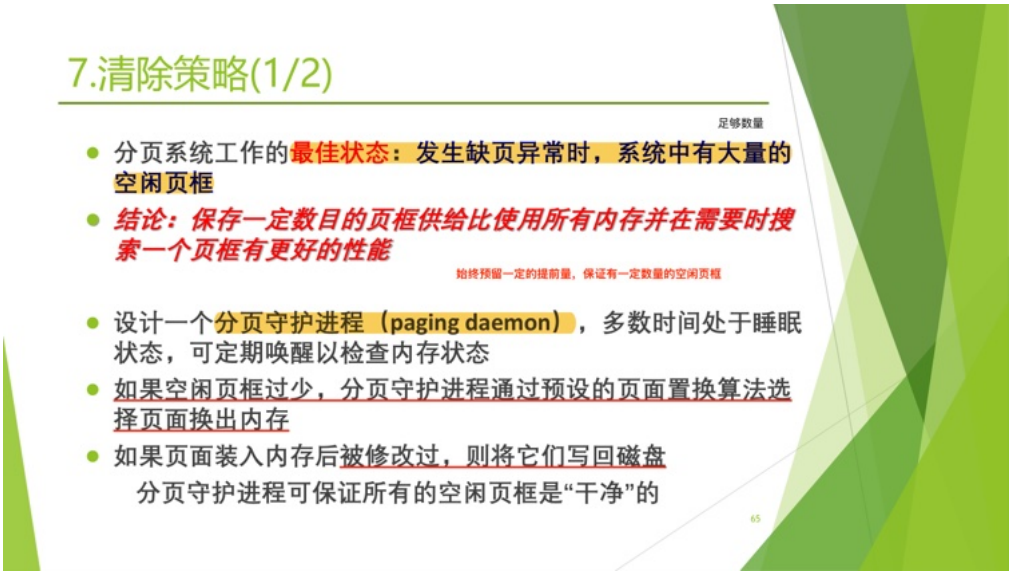


图示：OPT / NRU / LRU / FIFO / Second Chance / Clock / NFU / Aging / Working set / WSClock 的评价。

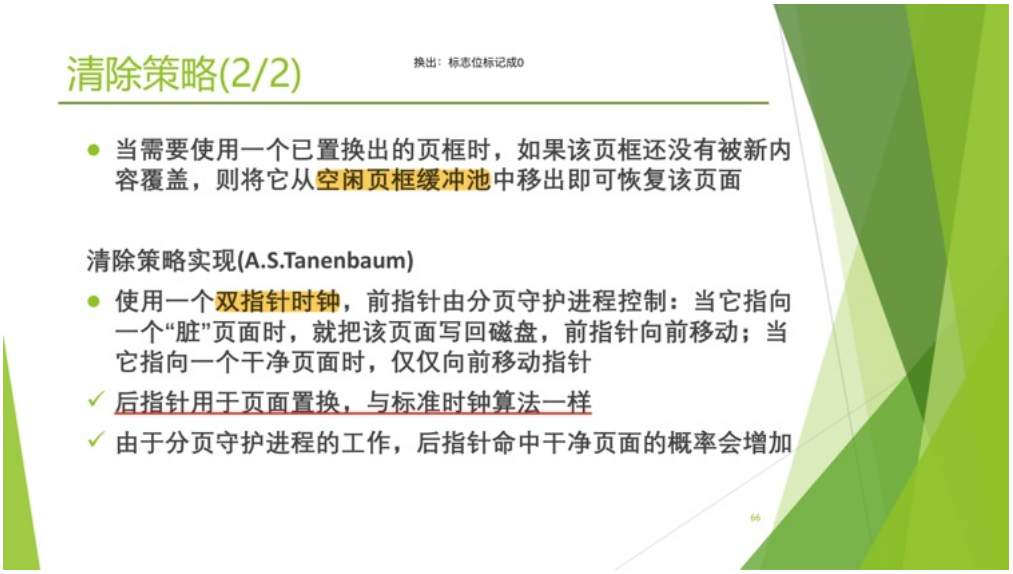
  - 后指针：与标准时钟一样进行置换；
- 由于守护进程的工作，后指针碰到干净页的概率显著上升，置换变快。

6.15.2 页缓冲技术

- 不立刻丢弃换出页，而是放入两个链表之一：
  - 未修改 -> 空闲链表；
  - 修改过 -> 修改链表；
- 修改链表以簇方式写回磁盘（一次写多个），减少 I/O 次数；
- 进程再次访问被换出页时，可直接从链表中拿回，几乎零代价（**软缺页**）。



图示：保留一定数量空闲页框；分页守护进程多数时间睡眠，定期检查并写回。



图示：用双指针时钟，前指针由分页守护进程控制写回，后指针进行置换。

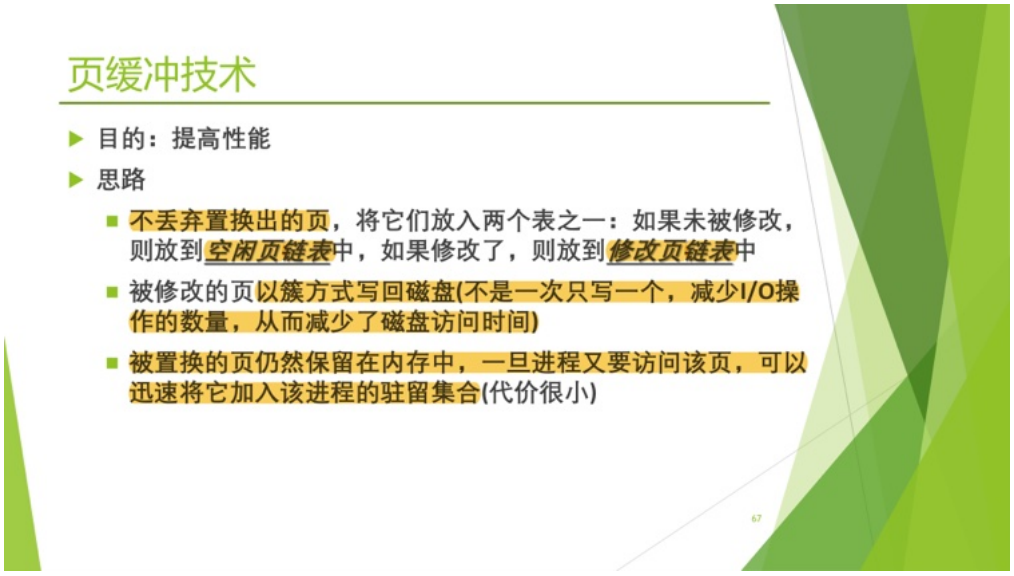
6.15.3 加载控制

- **系统并发度**：驻留在内存中的进程数目；
- 进程过多 -> 抖动；进程过少 -> CPU 利用率低；
- 解决方案：进程挂起，把它们的页换到磁盘；
- 选谁挂起需要考虑：低优先级、I/O 等待状态、近期未活动等。

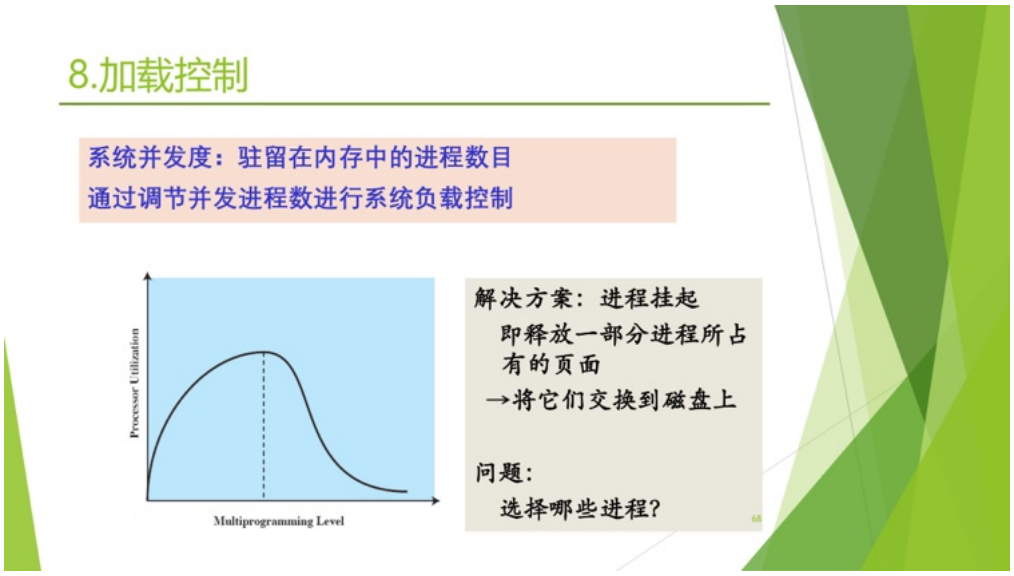
6.16 内存映射文件

6.16.1 基本思想

- 进程通过 `mmap` 系统调用把文件映射到虚拟地址空间的一个区域；



图示：置换出的页不立刻丢弃，放入空闲 / 修改链表；修改页以簇方式写回；常用页可被快速恢复。



图示：通过调节系统并发度（驻留进程数）控制内存压力；选择哪些进程挂起？

- 访问文件就像访问内存大数组，无需 read/write；
- 多数实现里，映射时不实际读入页面内容；首次访问触发缺页，按页读入；
- 进程退出或显式解除映射时，所有被修改的页面写回文件。

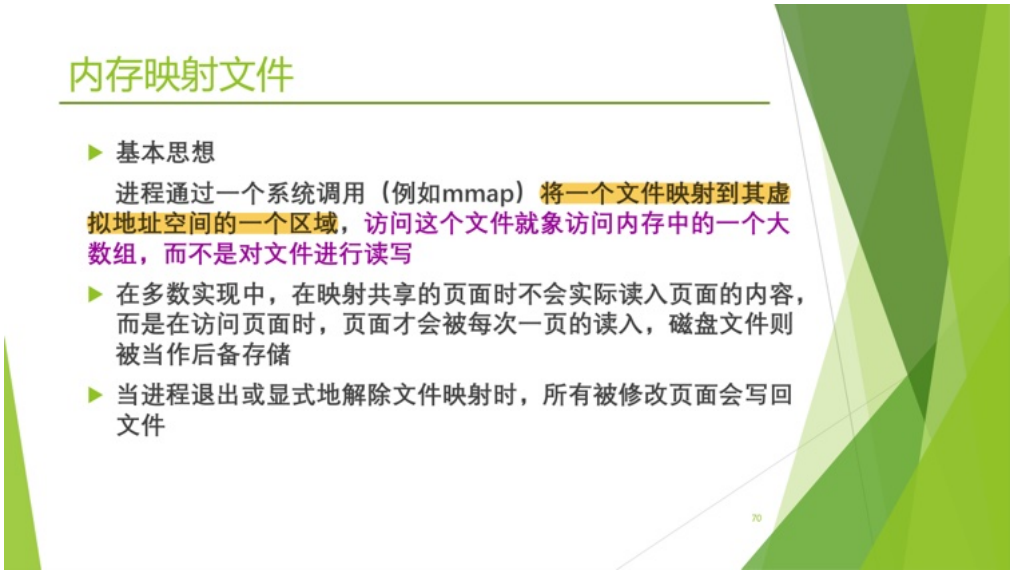
6.16.2 mmap 函数原型

参数解释：

- `prot`：页面访问权限位（PROT_READ / PROT_WRITE / PROT_EXE / PROT_NONE）；
- `flags`：映射对象类型（MAP_PRIVATE / MAP_SHARED / MAP_ANON 等）；
- `fd`：文件描述符；
- `offset` + `length`：从哪段开始、长度多少；



图示：mmap 是虚拟存储另一个重要应用。



图示：通过 mmap 把文件映射到地址空间；按页按需读入；解除映射时写回。

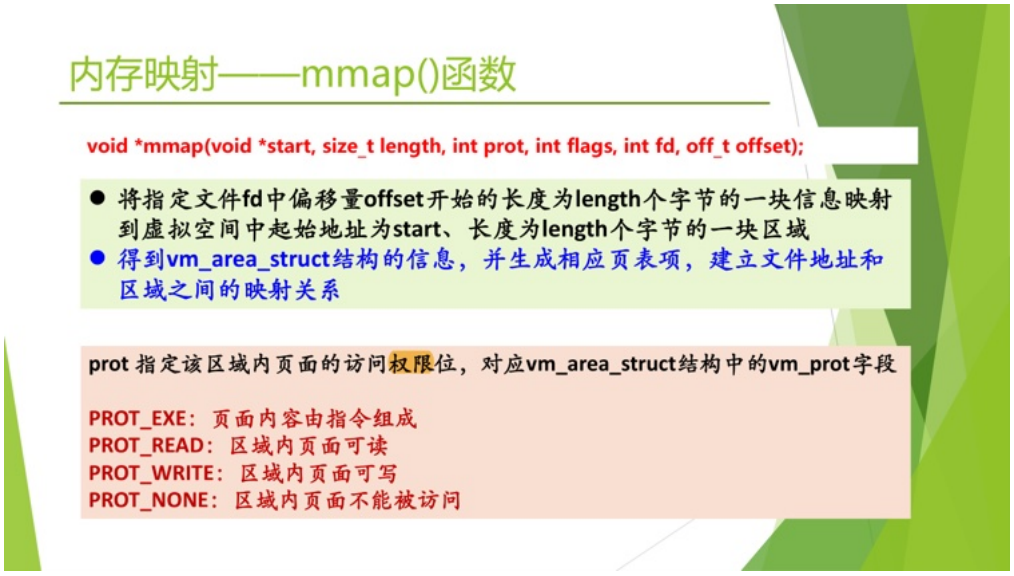
- 内核生成 `vm_area_struct` 并建立页表项，记录文件地址和虚拟区域的映射。

6.16.3 flags 详解

- `MAP_PRIVATE`:私有的写时复制对象;对应可执行文件中的 `.text / .rodata / .init / .data`;
- `MAP_SHARED`：共享对象；对应共享库或显式共享内存;
- `MAP_ANON`：请求 0 的页；对应内核创建的匿名文件;
- `MAP_PRIVATE | MAP_ANON`：未初始化数据（`.bss`）、堆、用户栈等。

6.16.4 虚拟存储管理全貌

把整张图记在脑里：



图示：`void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);`

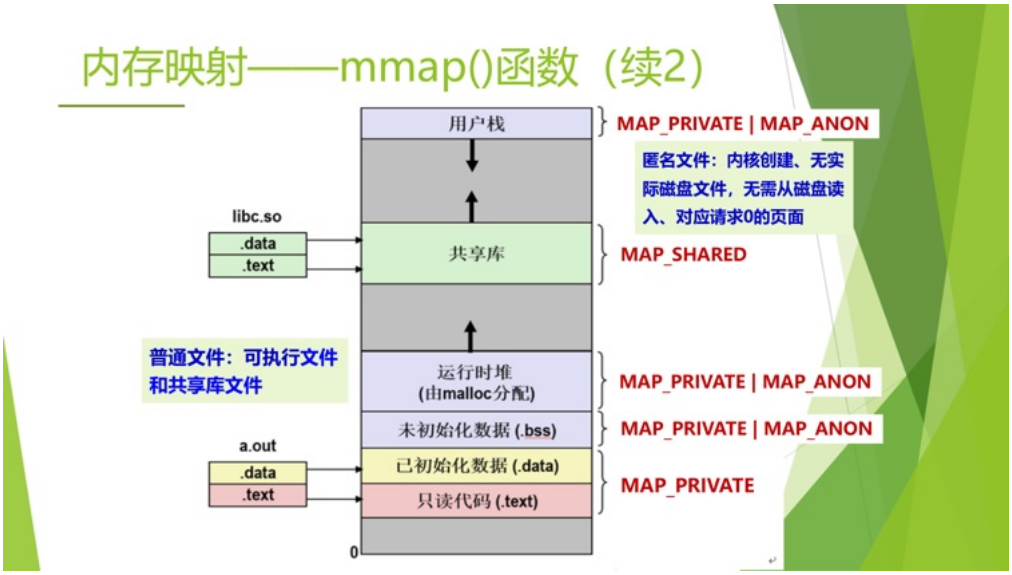


图示：MAP_PRIVATE / MAP_SHARED / MAP_ANON / MAP_PRIVATE | MAP_ANON 的语义。

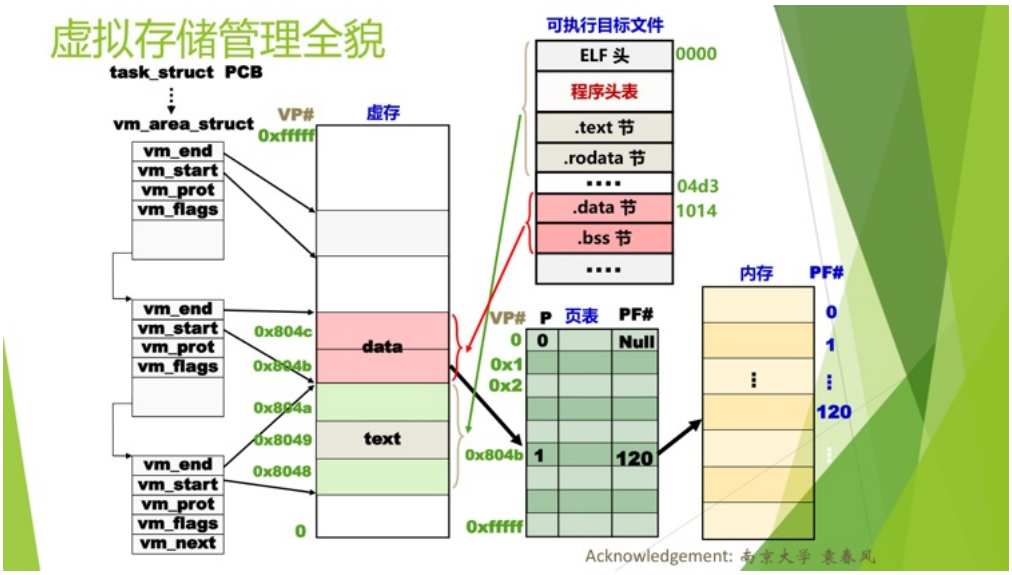
- `task_struct` 指向 `mm`；
- `mm` 维护一组 `vm_area_struct`；
- 每段 VMA 对应一个 `[start, end)` 虚拟范围、权限、可能映射到一个文件；
- 页表把虚拟页映射到物理页（PFN）；
- 文件 mapping 在“**未首次访问**”时通过缺页异常按需读入。

6.16.5 共享对象 vs COW 私有对象

- 多个进程都映射 libc 等共享库；
- 共享库代码在内存与磁盘各只需一份；
- 进程 1 启动时为共享对象分配若干页框；



图示：地址空间被一组 vm_area_struct 描述；对应的“文件”包括普通文件与匿名文件。



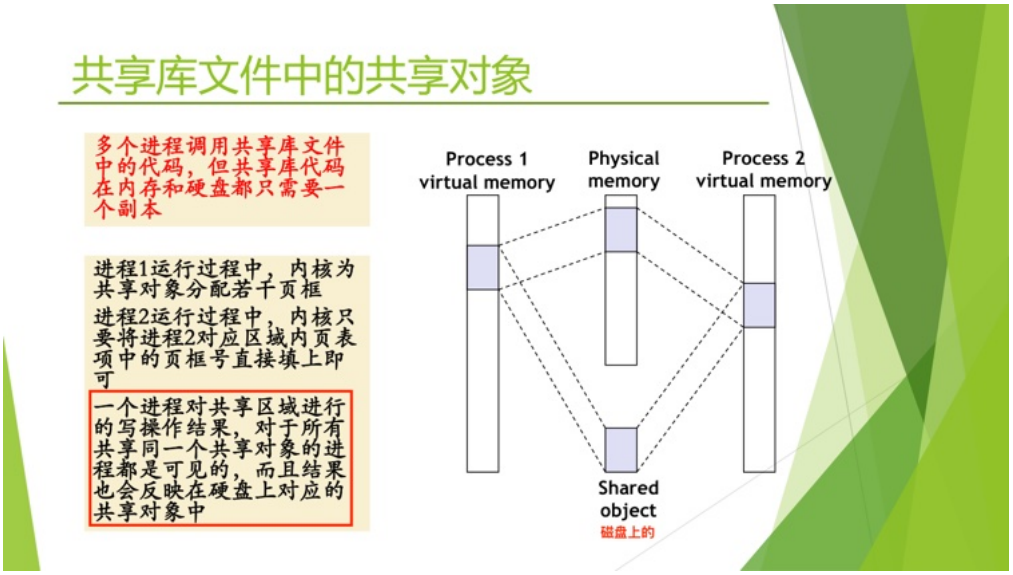
图示：task_struct -> mm -> vm_area_struct -> 虚拟地址段 -> 页表 -> 内存 -> 可执行目标文件。

- 进程 2 启动时只把页框号填到自己的页表项即可；
- 共享对象写操作对所有共享进程可见，最终也反映到磁盘文件。

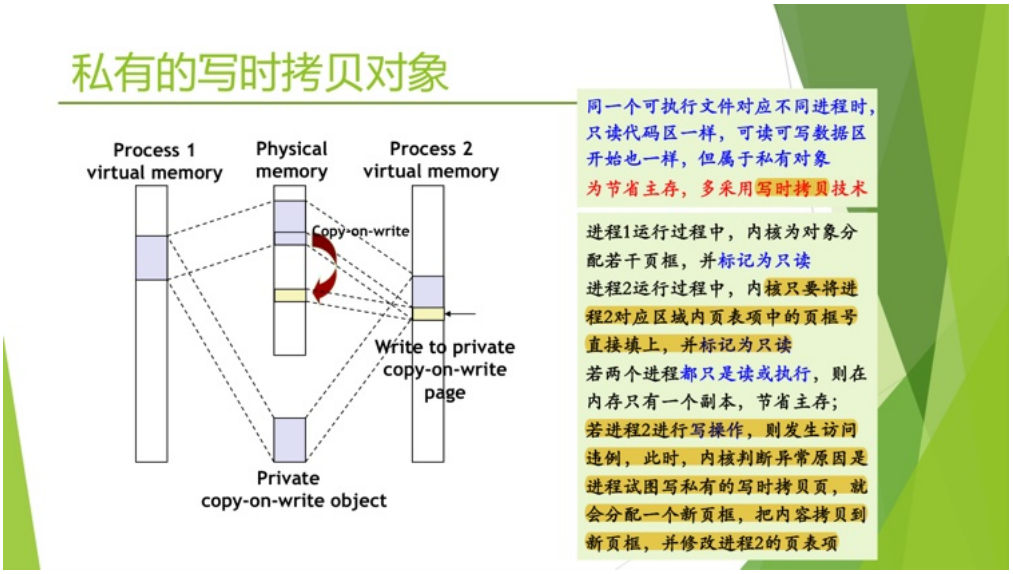
- 同一可执行文件对应不同进程时，只读区共享、可读写区初始也相同但属私有；
- 节省主存，多采用 COW；
- 进程 1 启动时分配若干页框、标记只读；
- 进程 2 启动时直接共享、也标记只读；
- 任意一方写入触发异常 -> 内核分配新页框 -> 复制并断开共享。

6.16.6 fork 与 COW

- 经典 `fork` 步骤：分配 proc / PID / 复制地址空间 / 继承共享资源 / 入就绪队列；



图示：多个进程映射同一个共享库，物理内存只保留一份。



图示：fork 后父子进程共享只读页；写时复制实现私有化。

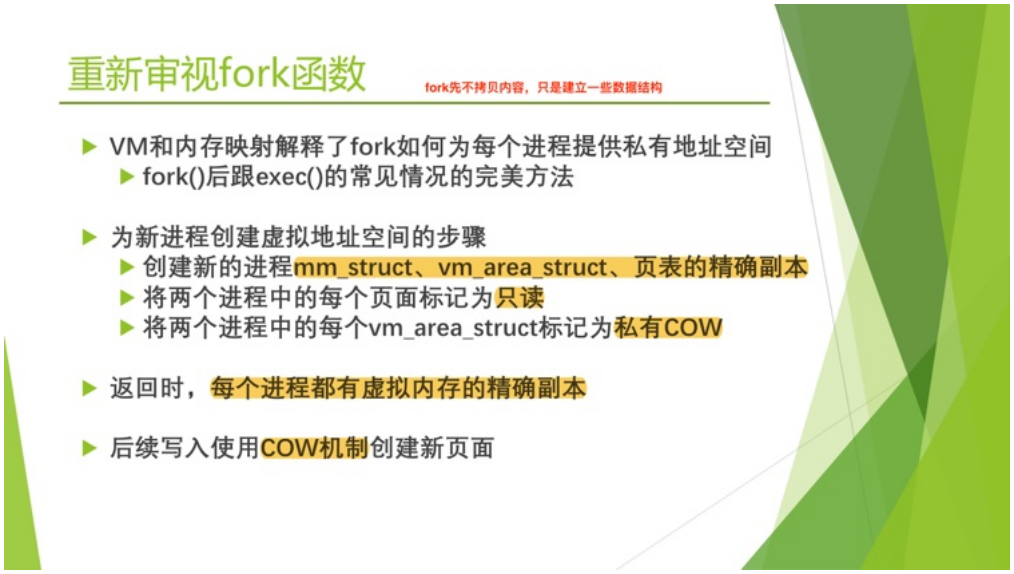
- COW 优化：
  - fork 不真的复制页内容，只复制数据结构（mm_struct、vm_area_struct、页表）；
  - 把页表项标记为只读；
  - 写入触发缺页异常时再复制页面。

6.16.7 内存映射文件的应用

- 数据库系统（LMDB）通过 mmap 把数据存放在文件中，访问像内存；
- 神经网络训练数据集（如 ImageNet）放在 LMDB；
- 高性能日志、缓存系统也常用 mmap。



图示：fork 经典实现 + Linux 用 COW 优化。



图示：fork 不立刻复制内容，只创建数据结构与只读页表项。

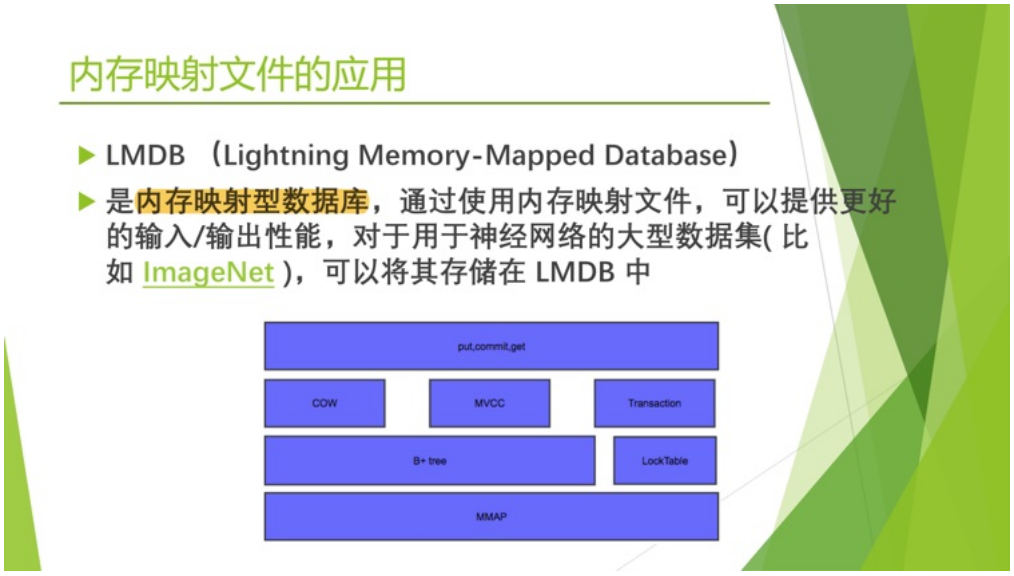
6.17 Windows 虚拟存储管理

6.17.1 Intel x86 虚拟内存机制

- x86 提供实模式、保护模式、长模式；
- 保护模式下用段选择符 + 段描述符做段式寻址；
- 段式映射后再做页式映射（控制寄存器 CR0 开启分页）。

6.17.2 内存管理器

- **执行体服务程序**：通过 Win32 API 或内核驱动接口暴露虚拟内存的分配、回收、管理。
- **缺页陷阱处理程序**：解决硬件检测的内存管理异常；负责分配页框、把磁盘数据装入内存。



图示：内存映射数据库 LMDB 把数据存放在 mmap 的文件中，提高 I/O 性能。



图示：Windows 虚存管理由内存管理器组、地址空间布局、地址转换、缺页处理等组成。

- **MmWorkingSetManager**：内存空闲低于阈值时启动所有内存管理策略，包括工作集修整、老化、写回脏页等；
- **KeSwapProcessOrStack**：完成进程和内核线程堆栈的换入换出；
- **MiModifiedPageWriter**：把脏页写回页文件；
- **MiMappedPageWriter**：把映射文件中的脏页写回磁盘；
- **MmZeroPageThread**：把空闲链表中的页框清零。

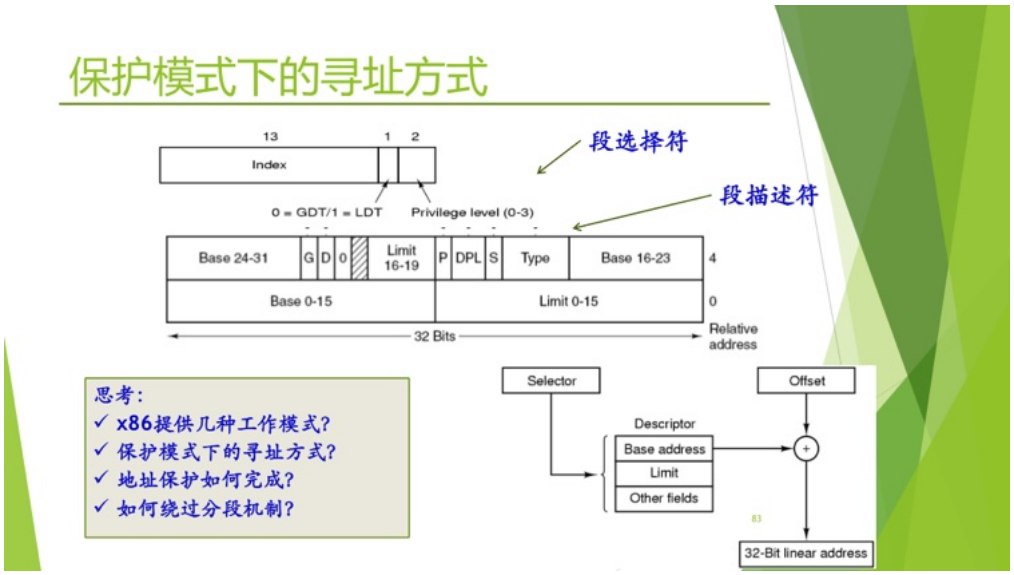
6.17.3 地址空间布局与磁盘后备

6.17.4 地址转换机制

- 每个进程有自己的页目录与页表（私有部分）；



图示：段式 + 页式；段描述符；地址转换。

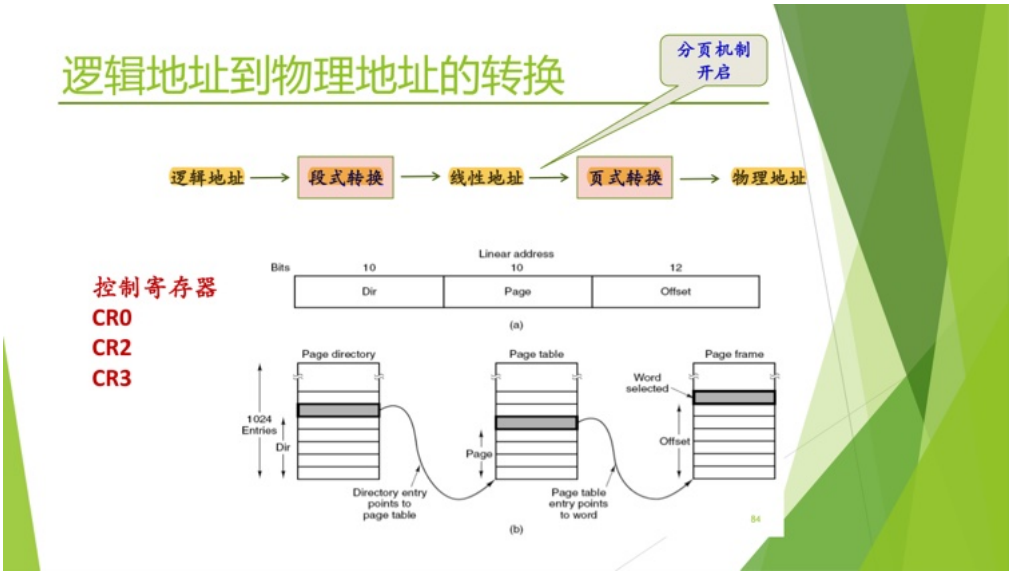


图示：x86 工作模式 / 段选择符 / 段描述符 / 地址保护 / 绕过分段机制。

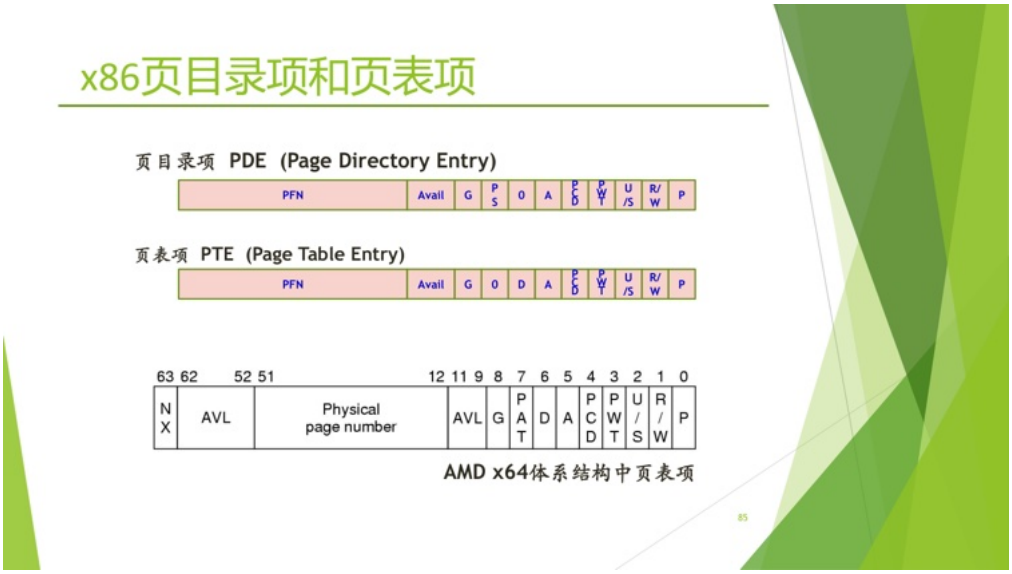
- 系统页表对所有进程公共，描述内核空间；
- 用户切换时只需切换 CR3 指向的进程页目录。

6.17.5 缺页异常处理

- 所引用的页面没有提交（不在内存）；
- 违反权限的页面访问；
- 修改一个私有共享 COW 页面；
- 需要扩展栈；
- 所引用的页已提交但尚未映射；
- 请求一个 0 页面。



图示：CR0 控制分页开启；CR2 缺页地址；CR3 页目录基址；逻辑地址 -> 段式 -> 线性地址 -> 页式 -> 物理地址。

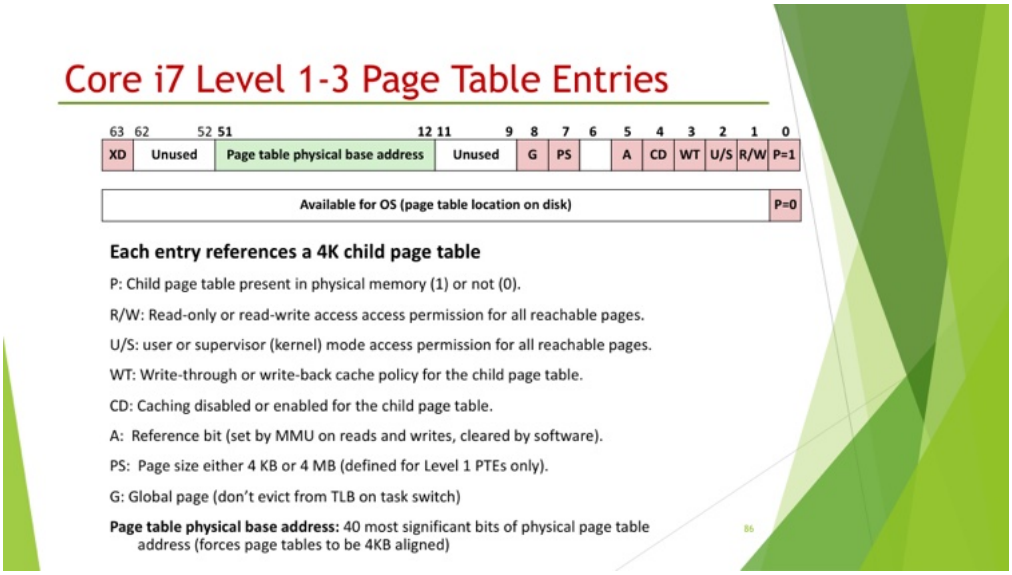


图示：x86 32 位页式分页机制中页目录项与页表项的位格式（与 §4.1 相同）。

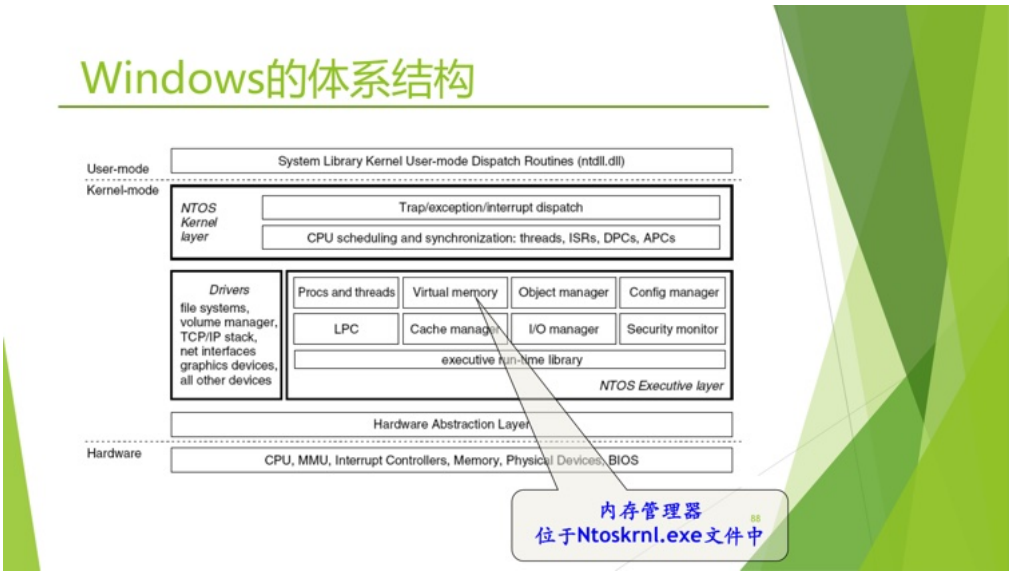
- CPU 触发缺页时把虚拟地址放进 CR2；
- 中断号 0xe；
- IDT 中第 0xe 项指向 `ntoskrnl!KiTrap0E`；
- KiTrap0E 调用 `MmAccessFault`；
- MmAccessFault 通过 CR2 计算 PDE/PTE，分析 PTE 内容来决定如何处理。

6.17.6 页目录自映射机制

- 页目录是一个内存中的特殊页，物理地址保存在 KPROCESS；
- 硬件用 VPN/PFN 访问页目录、页表与页；
- 内核需要用**虚拟地址**访问页表，因此把页目录映射到固定虚拟地址（x86 上是 0xC0300000）；



图示：Core i7 多级页表项的具体字段含义。



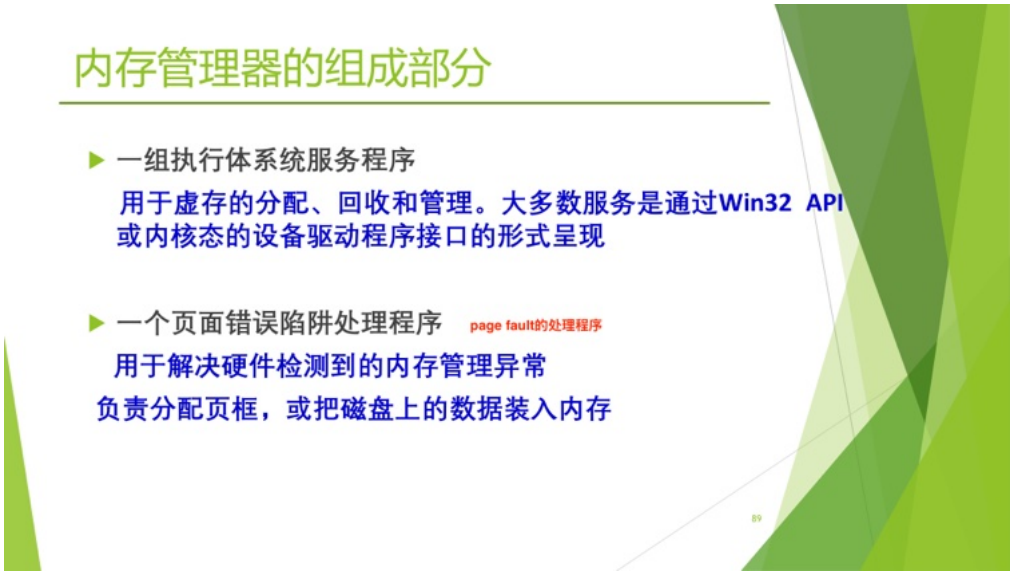
图示：内存管理器位于 Ntoskrnl.exe，是执行体的一部分。

- 这就是“**自映射**”——页目录中第 0x300 项指向自己。

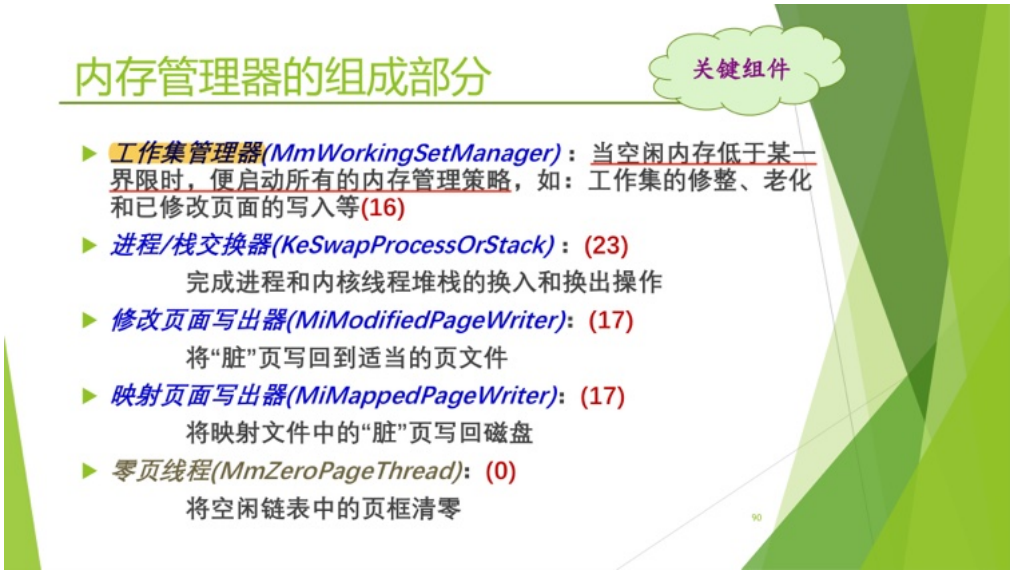
6.17.7 工作集

- **进程工作集**：分给每个进程的若干页框；
- **系统工作集**：分给可分页系统代码和数据；
- 内存管理器使用请求式调度 + 簇方式装入；
- 置页策略让 cache 不必要的震荡最小。

6.17.8 用户空间内存分配方式



图示：执行体服务、缺页陷阱处理程序、各种线程。

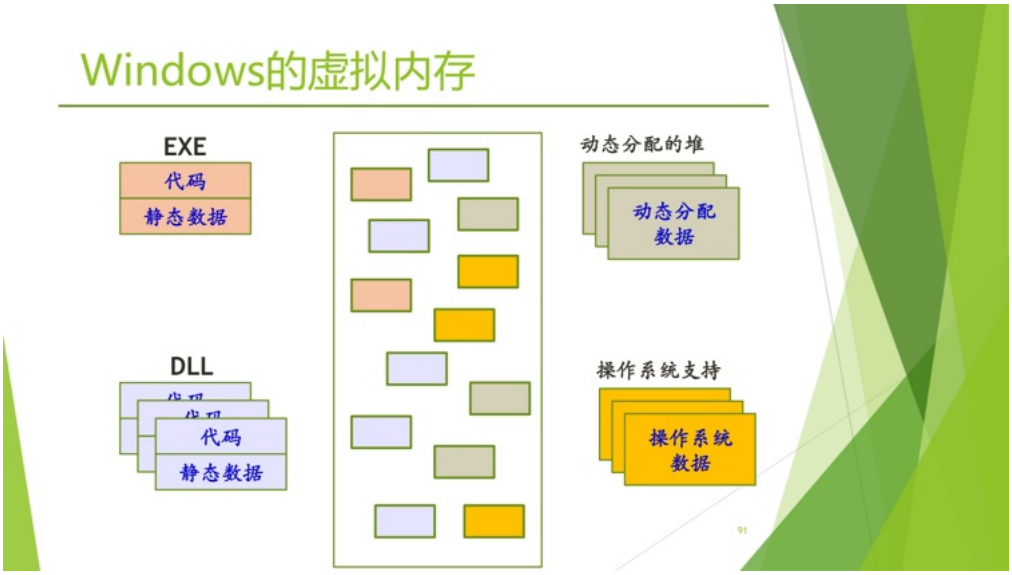


图示：MmWorkingSetManager / KeSwapProcessOrStack / MiModifiedPageWriter / MiMappedPageWriter / MmZeroPageThread。

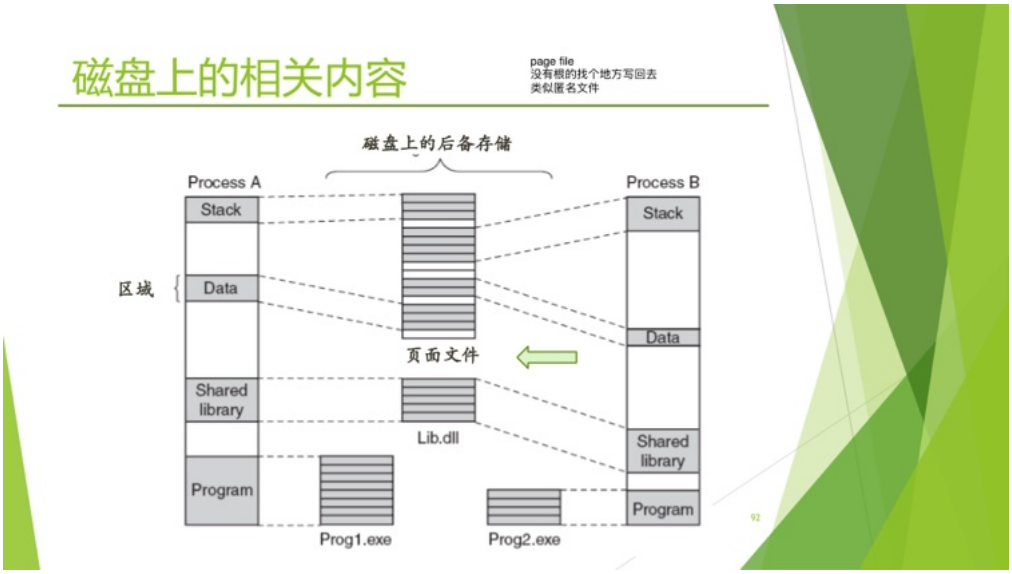
VirtualAlloc

- VirtualAlloc 把页虚拟内存“保留”和“提交”分开：
  - 保留：占住一段虚拟地址但不分配实际物理页；
  - 提交：在保留的虚拟范围中真正分配物理页框；
- VirtualFree / VirtualFreeEx 回收页面或释放地址空间；
- 适合需要稀疏地址布局或预先保留地址的场景。

- 提交可以靠页表判断（PTE 是否有效）；
- 但保留无法靠页表区分；
- 因此 Windows 引入 VAD 来记录每段虚拟空间的保留 / 提交状态。



图示：进程地址空间分为代码、静态数据、动态数据、堆、操作系统空间；OS 在每个进程地址空间高位。

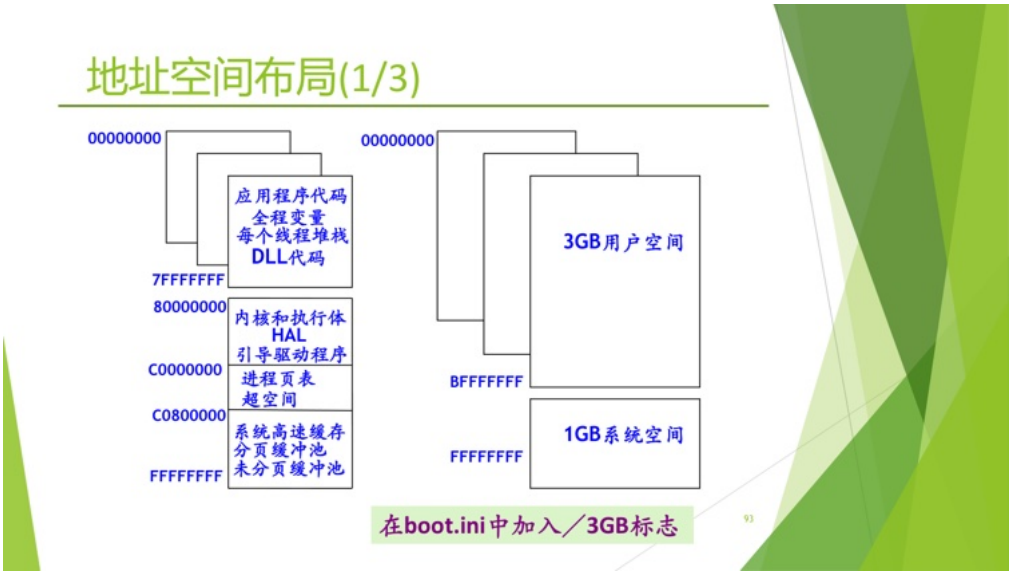


图示：page file 是 Windows 的“匿名 swap”文件，存放无文件后备的虚拟页。

- VAD 类似 Linux 的 `vm_area_struct`；
- 二叉排序树根挂在 EPROCESS 中。

内存映射文件

- `CreateFileMapping` 创建区域对象（section object）；
- `MapViewOfFile` 把区域对象的一部分映射到地址空间；
- 多个进程对同一区域对象建立视图就构成共享；
- 访问时若缺页，由内存管理器自动从文件读入；
- 修改后通过 `FlushViewOfFile` 显式刷新或常规调度时写回。



图示：3GB 用户空间 + 1GB 系统空间；通过 boot.ini /3GB 标志可将用户空间扩为 3GB。



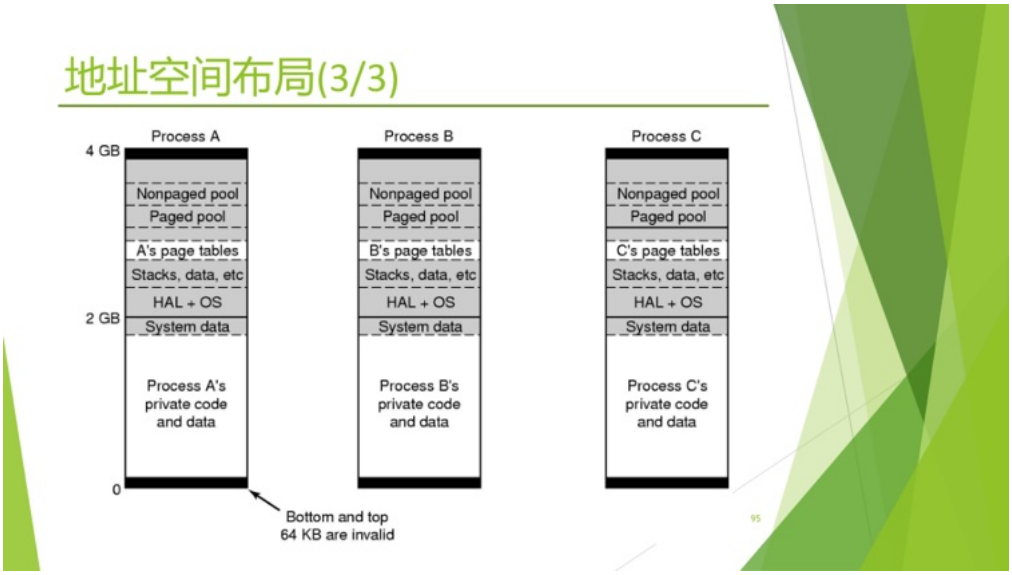
图示：系统代码、系统映射视图、超空间、进程页表、附加 PTE、未使用、HAL、故障转储、未分页缓冲池扩展、系统 PTE、分页缓冲池、系统高速缓存、系统工作集列表。

内存堆

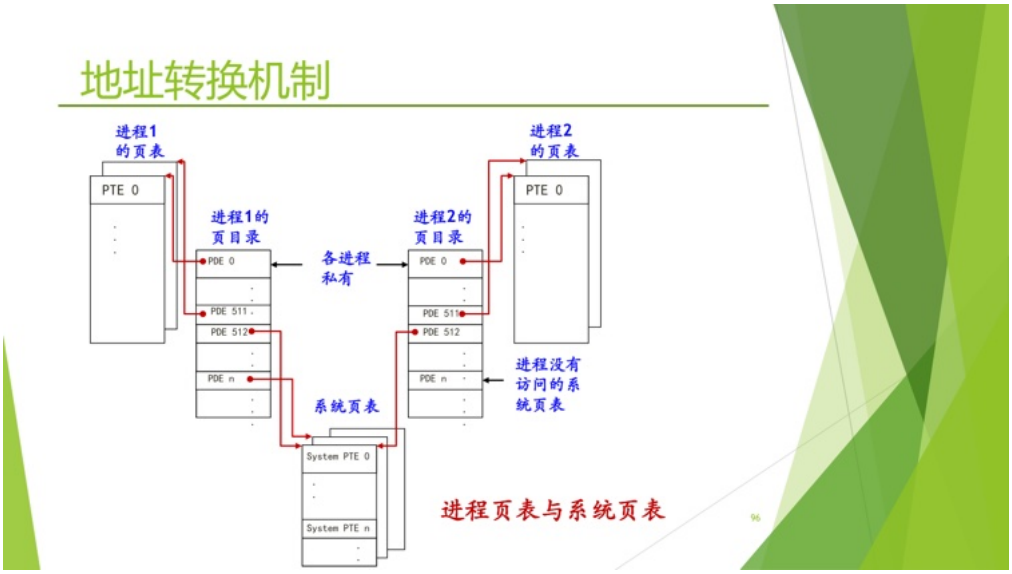
- 堆是保留地址空间中一个或多个页组成的区域，由堆管理器细分小块；
- 进程启动时带一个默认进程堆（通常 1MB）；
- 通过 HeapAlloc / HeapFree 在堆中分配 / 释放小块；
- 进程也可创建私有堆（HeapCreate）；
- 堆管理器是执行体中分配 / 回收可变内存的函数集。

6.17.9 物理内存管理

- Windows 维护一个 PFN 数据库；
- 每个物理页对应一个 MMPFN 项（24 字节）；



图示：各区域用途总览。

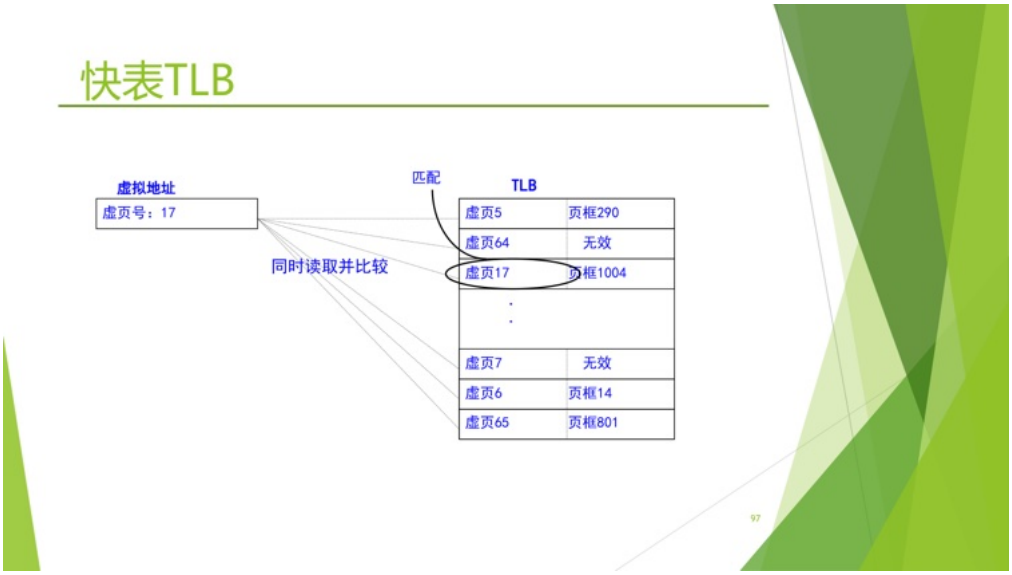


图示：每个进程有自己的页目录与页表，系统页表对所有进程公共。

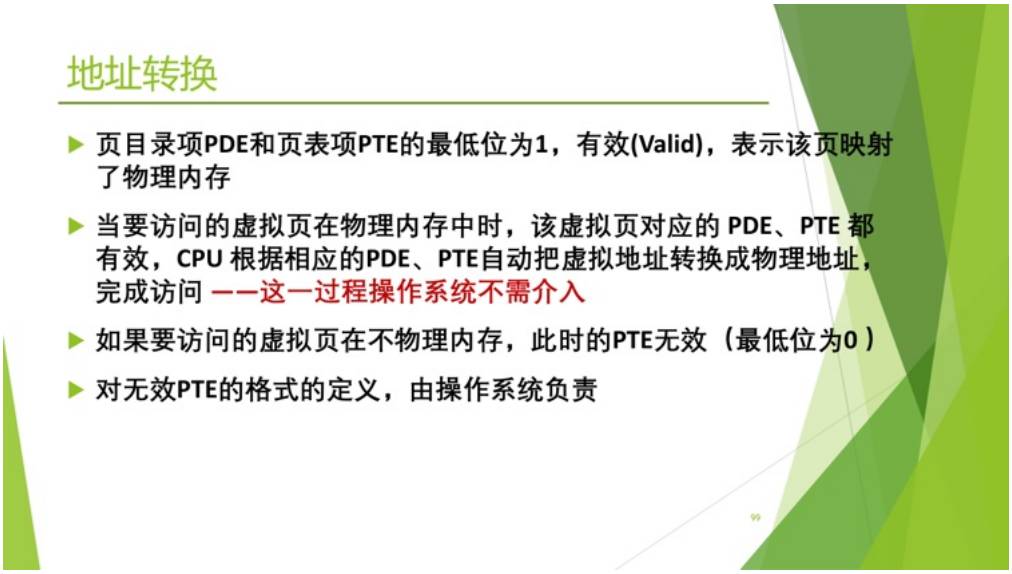
- 全局变量 MmPfnDatabase 是数据库首地址；
- PFN 项记录每页的所有元数据（状态、引用计数、共享情况等）。

页框状态

- **活动（Active / Valid）**：在某进程工作集中，PTE 有效；
- **过渡（Transition）**：正在从文件读入或写回；
- **空闲（Free）**：内容不再被需要；
- **零初始化（Zeroed）**：空闲且已被清零；
- **坏（Bad）**：硬件错误，不能使用；
- **修改（Modified）**：曾在工作集，被改过；
- **后备（Standby）**：曾在工作集，但未被修改。



图示：通过 TLB 的相联存储器并行比较所有项，匹配时直接取出页框。



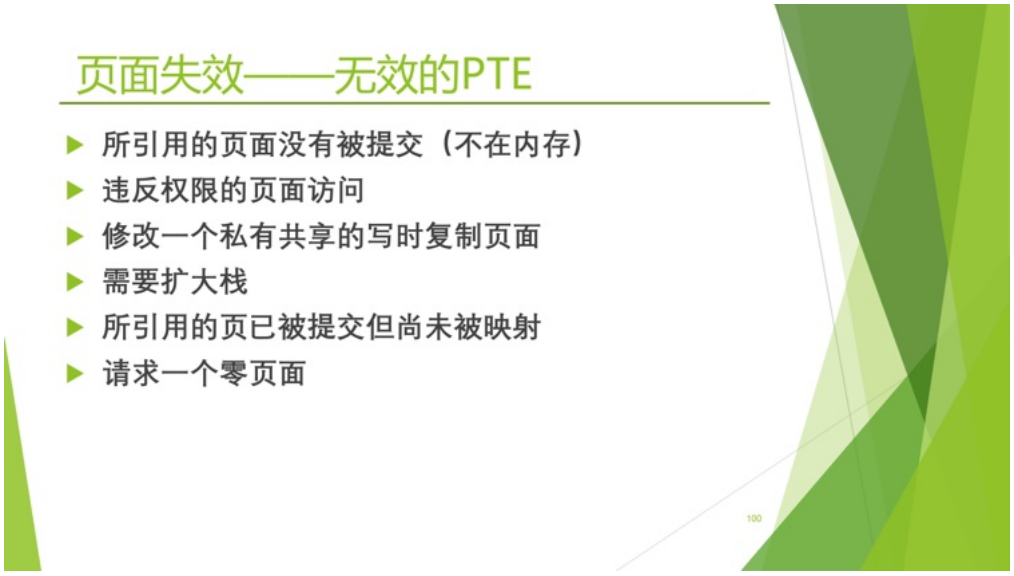
图示：PDE/PTE 的最低位为 1 表示有效；为 0 表示无效，由 OS 定义。

- 修改 / 后备页都仍在内存，可在再次访问时快速恢复成 Active；
- Modified 页在被作其他用途前需要写回页文件。

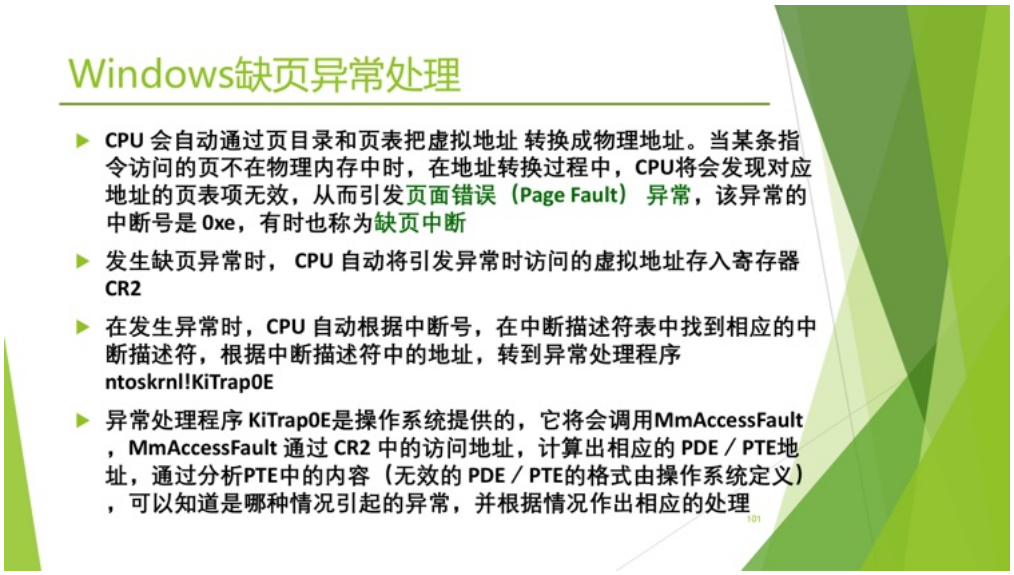
整体协作

7 步要点：

1. **释放页面**：`VirtualFree` / 进程终止 -> 直接进入空闲链表；
2. **硬缺页 / 磁盘读入**：访问不在内存的页 -> 从空闲链表取一页框，从磁盘读入，加入工作集；
3. **工作集置换**：内存压力下把较久未用的页移出工作集；干净页进 Standby，脏页进 Modified；
4. **修改页写出**：MiModifiedPageWriter 把 Modified 写回页文件，写完进入 Standby；
5. **后备命中**：进程再次访问 Standby 页 -> 软缺页异常 -> 直接拉回工作集；



图示：未提交、违权、私有共享 COW、扩栈、已提交但未映射、需求 0 页面。



图示：CR2 保存触发地址 -> IDT 第 0xe 号 -> ntoskrnl!KiTrap0E -> MmAccessFault -> 处理。

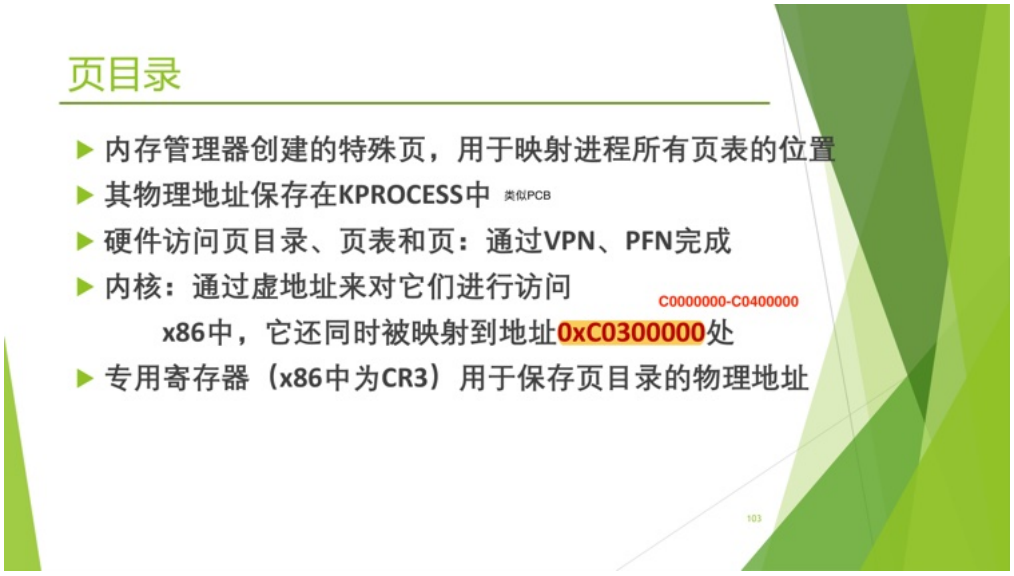
6. **零初始化线程**：把 Standby / Modified 中已写回的页清零，进入零初始化链表；
7. **需求零缺页**：进程访问匿名页 -> 从零初始化链表分配，避免再清零。

写时复制示例

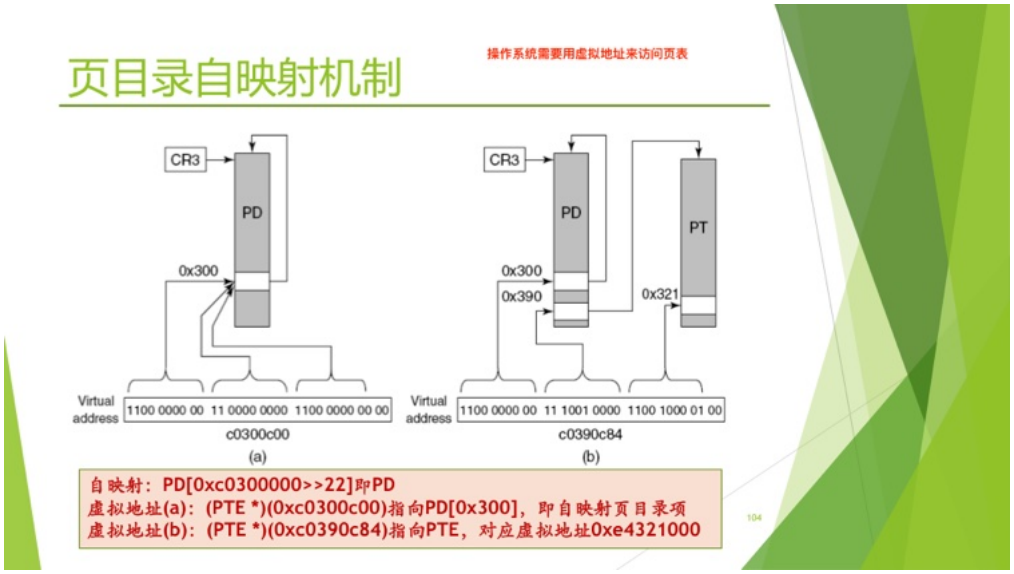
- 两个进程映射同一区域，三个页都标 COW；
- 进程 1 写第 2 页：分配新物理页，复制内容，进程 1 的 PTE 改为指向新页；
- 新副本只对进程 1 可见，其他进程仍共享原始页。

6.18 重点小结

记住几条主线：

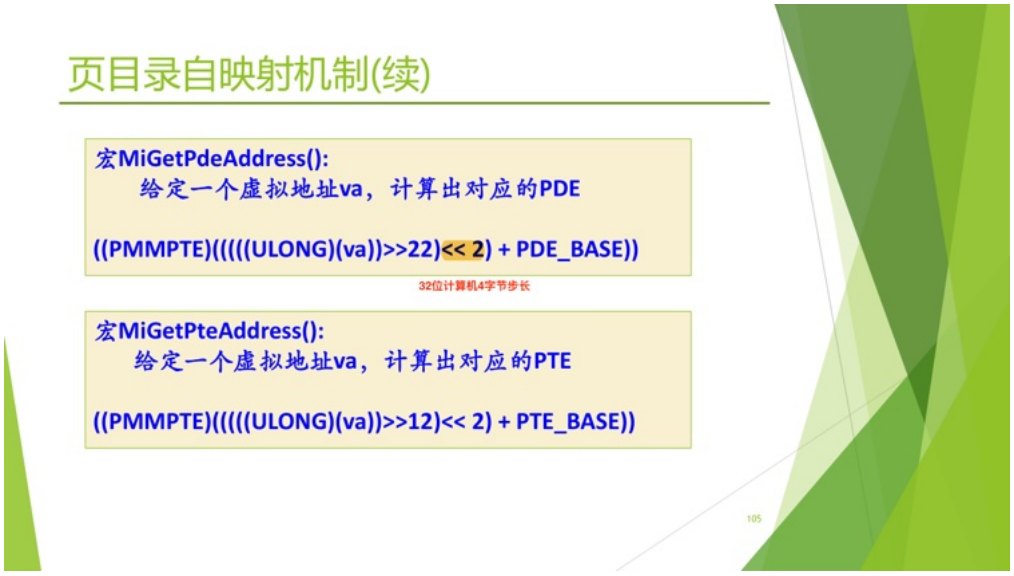


图示：页目录是一个特殊页，页目录项指向自身；用虚拟地址访问页表。

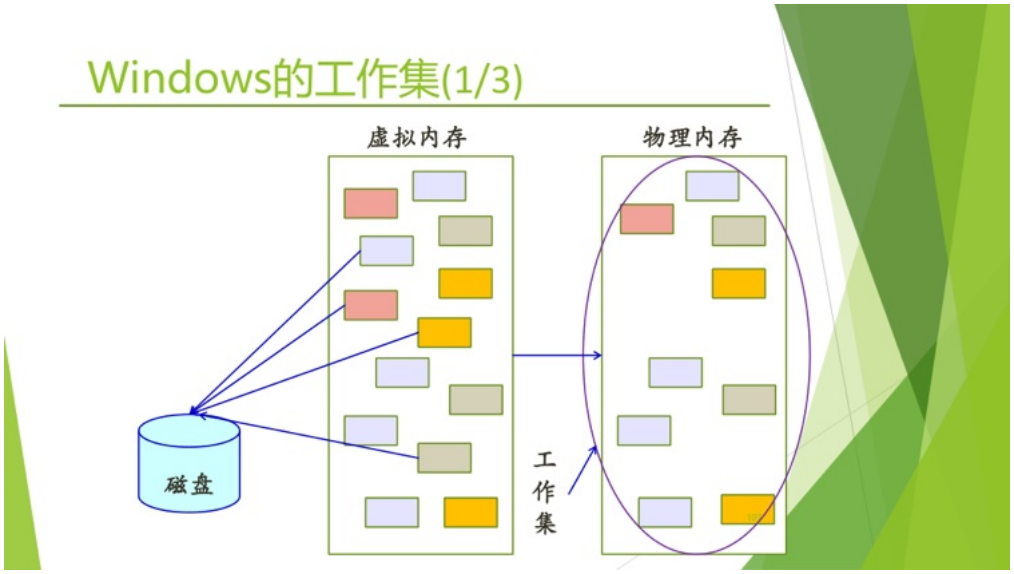


图示：通过 PD[0xC0300000»22] = PD 实现自映射；演示如何用宏取得任意虚地址对应的 PDE / PTE。

- **基础**：虚拟内存 / 虚拟地址 / 虚拟地址空间 / 页表 / TLB / 反置页表；
- **硬件机制**：地址转换、PTE 控制位、缺页异常入口；
- **软件策略**：取页（按需 / 预取）、放置、置换（局部 / 全局）、驻留集（固定 / 可变）、清除（分页守护进程 / 页缓冲）、加载控制（挂起进程）；
- **算法谱系**：OPT / FIFO / SCR / Clock / NRU / LRU / NFU / Aging / Working Set / WSClock；
- **特殊机制**：内存映射文件、写时复制、共享对象、自映射页目录；
- **真实系统**：Windows 内存管理器（PFN 数据库 / VAD / 工作集修整）。



图示：MiGetPdeAddress / MiGetPteAddress 计算给定虚拟地址对应的 PDE 与 PTE 在虚拟空间中的地址。



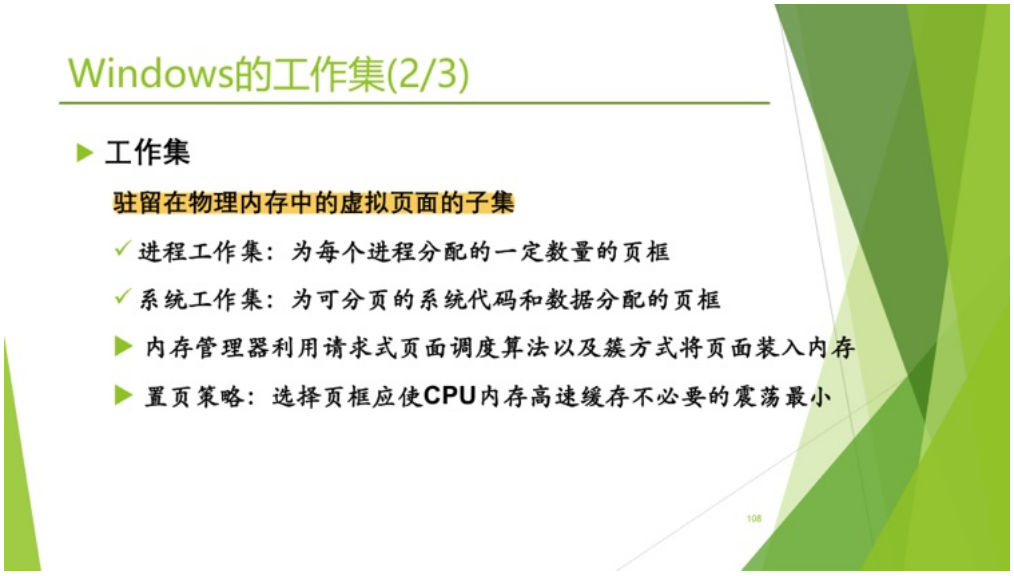
图示：进程虚拟内存 -> 物理内存 -> 磁盘；驻留物理内存的虚拟页面集合就是工作集。

6.19 XV6 源代码阅读

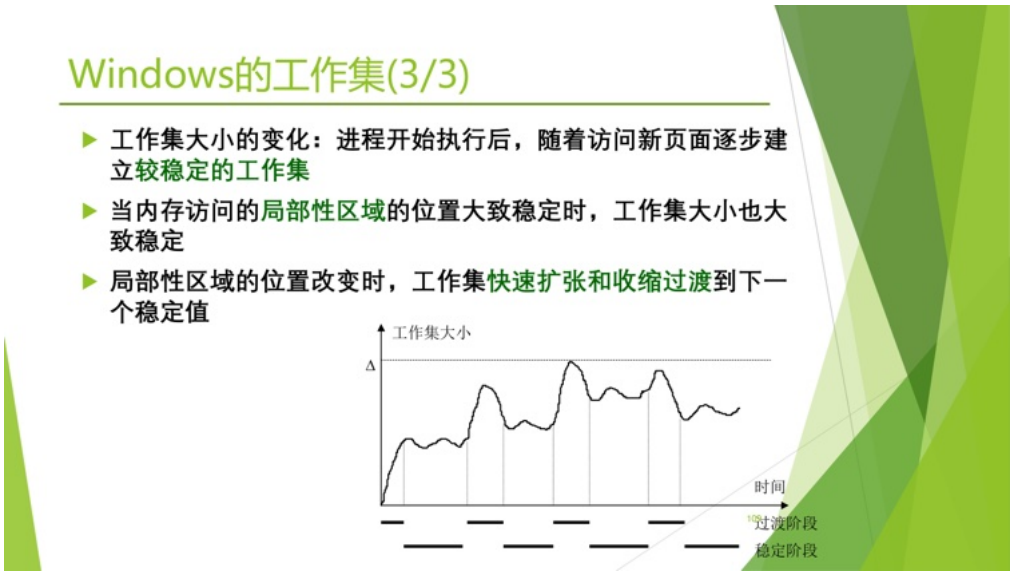
> **代码阅读题：**阅读 XV6 中 `kernel/memlayout.h`, `vm.c`, `kalloc.c`, `exec.c`, `elf.h`, 说明 XV6 的虚拟存储管理。

解答：

1. `memlayout.h`：
   - 定义内核与用户态的关键虚拟地址常量：`KERNBASE`, `PHYSTOP`, `TRAMPOLINE`, `TRAPFRAME`；
   - 用户态地址从 0 开始；trampoline 与 trapframe 在用户与内核中都映射在最高地址，确保陷入 / 返回路径中页表切换的连续性。
2. `kalloc.c`：



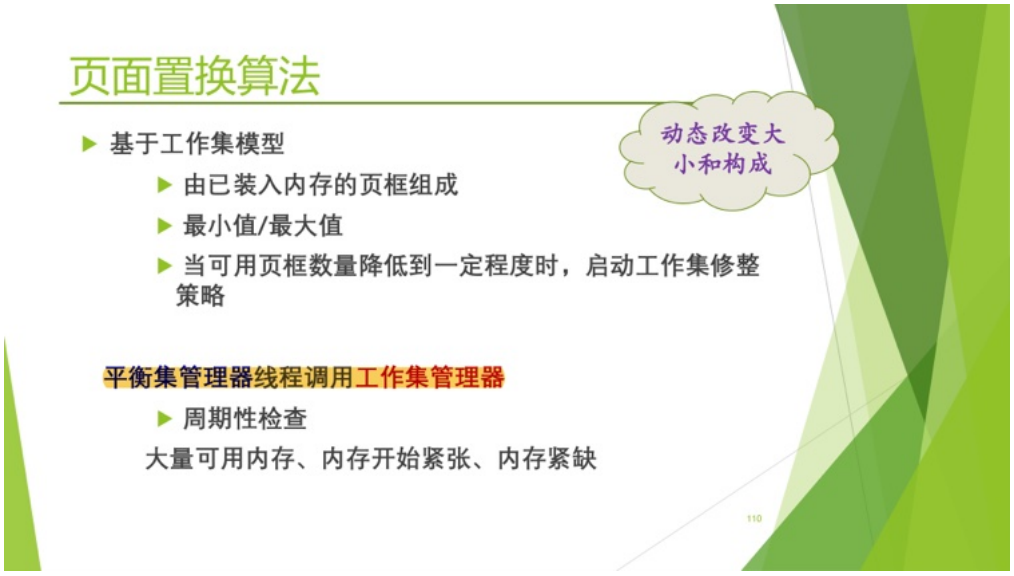
图示：进程工作集与系统工作集；置页策略尽量减少 cache 震荡。



图示：进程开始执行后工作集逐步建立 -> 稳定 -> 局部性变化时再次过渡。

- XV6 的物理页分配器；
- 用一个空闲页链表管理 4KB 物理页；
- `kinit` 把所有可用物理页加入链表；
- `kalloc` 取链表头返回一个空闲页；`kfree` 把一页归还。

3. `vm.c`：
   - 维护页表；
   - `walk(pagetable, va, alloc)` 沿三级页表查找 PTE，可选自动分配中间级页表；
   - `mappages` 在指定虚拟范围内为每页建立 PTE；
   - `uvmalloc` 为用户进程扩展地址空间，分配页并 mappages；
   - `uvmcopy` 复制父进程页表到子进程（XV6 简单实现，没有 COW）；
   - `uvmunmap` 取消映射并释放页。



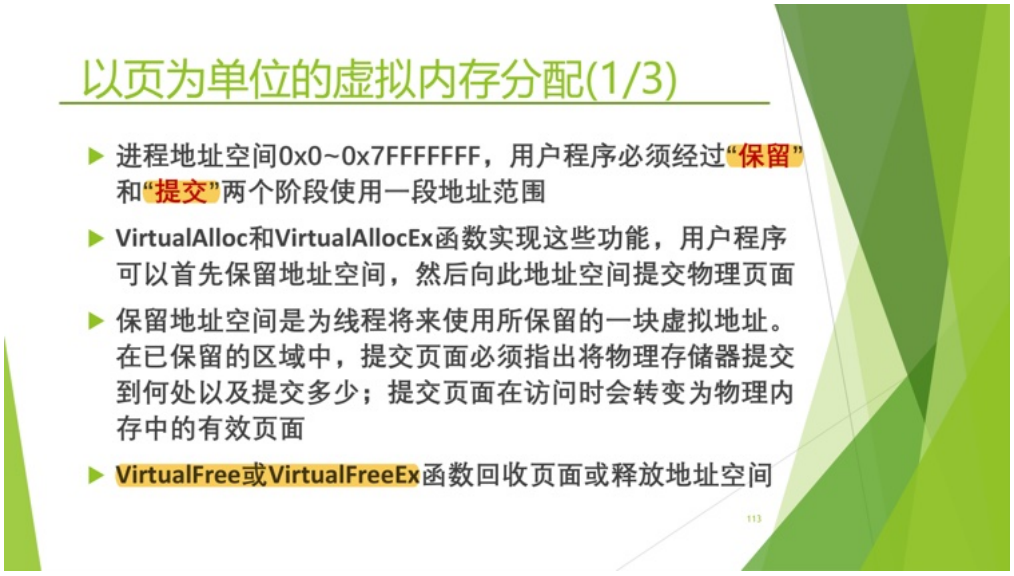
图示：最小值 / 最大值；可用页框数下降时启动工作集修整；周期性检查。



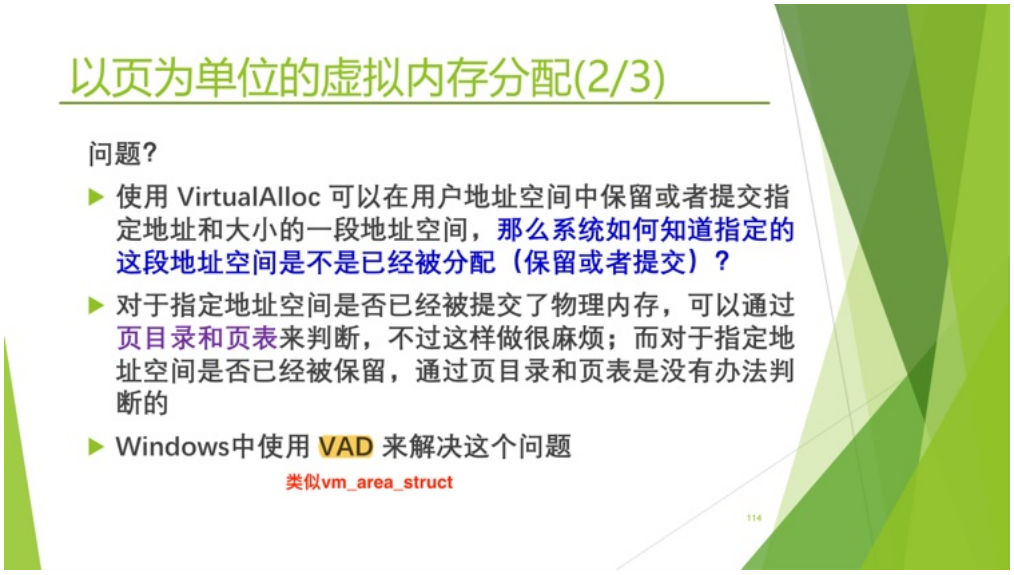
图示：以页为单位的虚拟内存分配（VirtualAlloc）/ 内存映射文件（CreateFileMapping）/ 堆（HeapAlloc）。

4. `exec.c`：
   - 加载 ELF 可执行文件：
     - `namei` 取得文件 inode；
     - 解析 `elfhdr`（来自 `elf.h`），按程序头逐段加载到新的页表中；
     - 用 `uvmalloc` 分配栈，把命令行参数构造在用户栈顶；
     - 替换 `proc->pagetable`、`proc->sz`、`proc->trapframe` 等；
   - 返回到 user space 时已运行新程序。
5. `elf.h`：定义 ELF 的 magic、`elfhdr`、`proghdr` 等结构，是加载可执行文件的格式描述。

整体看下来，XV6 的虚拟存储是 §3 ~ §10 内容的最小可运行实现：硬件机制（页表 / 三级查表 / 缺页）、空闲物理页管理（kalloc）、地址空间装载（exec）、`vm.c` 中各类对地址空间的操作。理

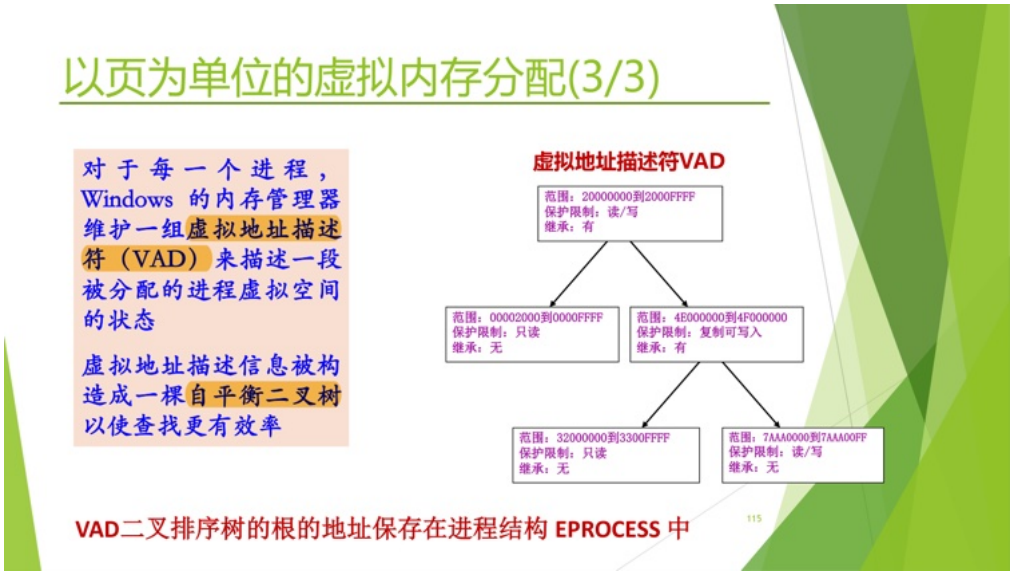


图示：进程地址空间 0x0–0x7FFFFFFF；通过“保留”和“提交”两步使用一段地址范围。

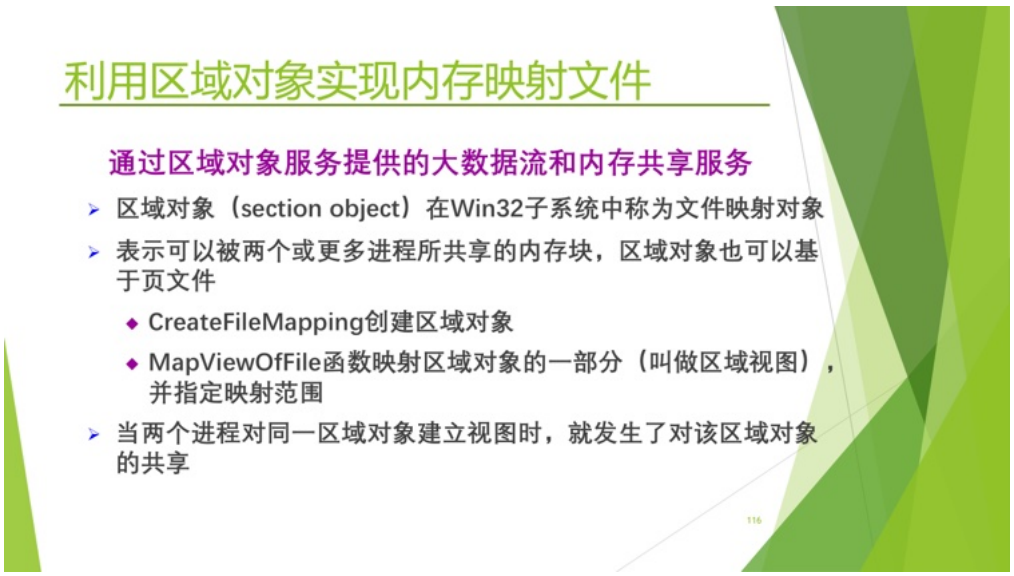


图示：判断地址是否已被保留 / 提交需要 VAD（虚拟地址描述符），不能仅靠页表。

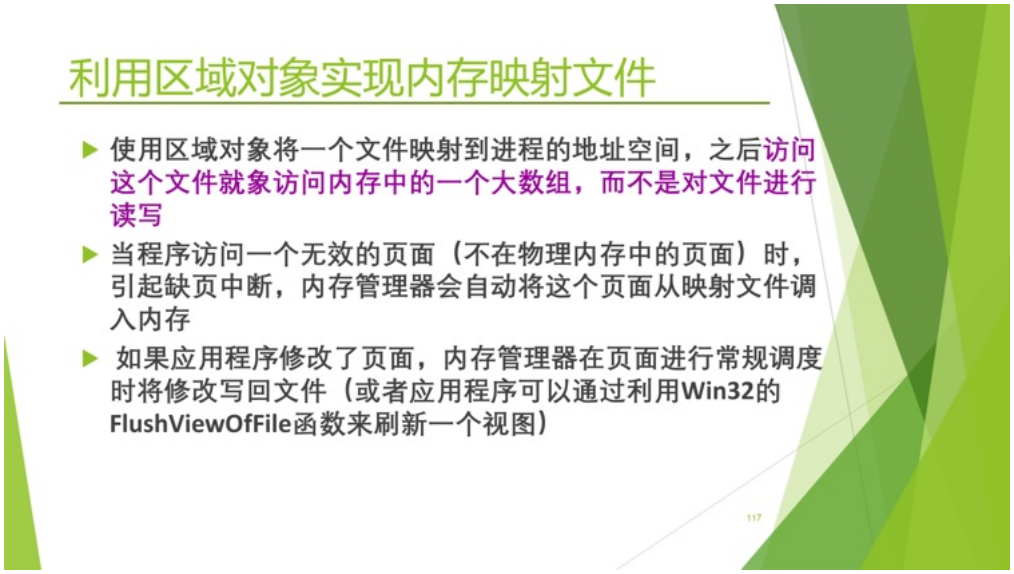
解了 XV6 这一套，就能更好地阅读 Linux 中复杂得多的虚拟存储实现。



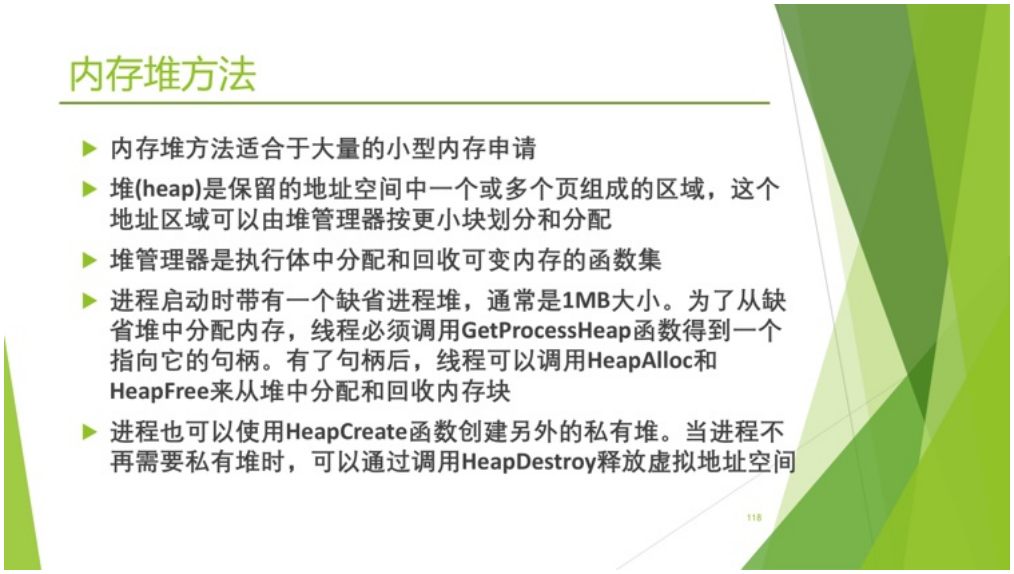
图示：VAD 描述每段被分配虚拟空间的范围、保护、继承属性；构造为自平衡二叉树以加快查找。



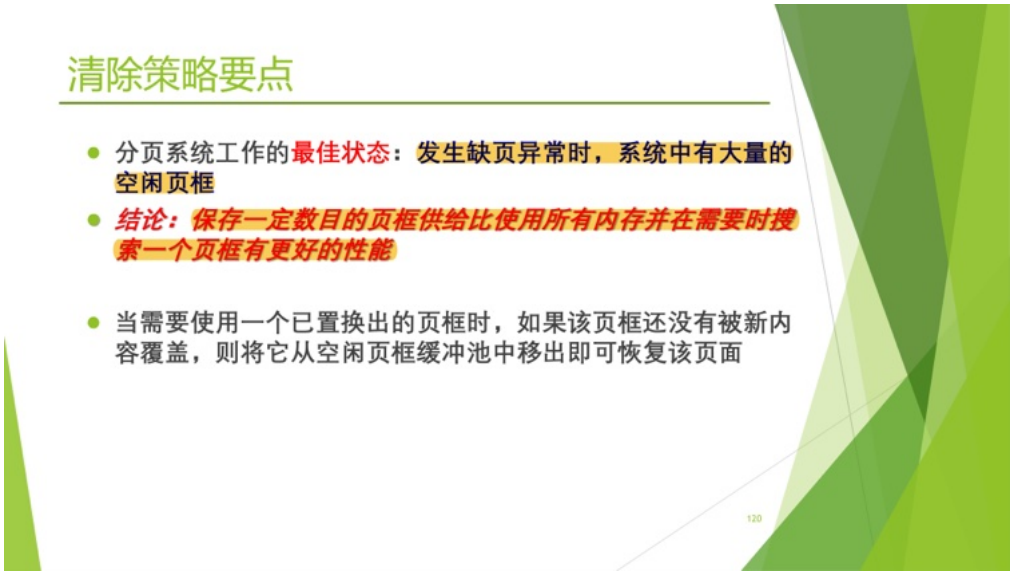
图示：区域对象在 Win32 中称为文件映射对象；CreateFileMapping、MapViewOfFile。



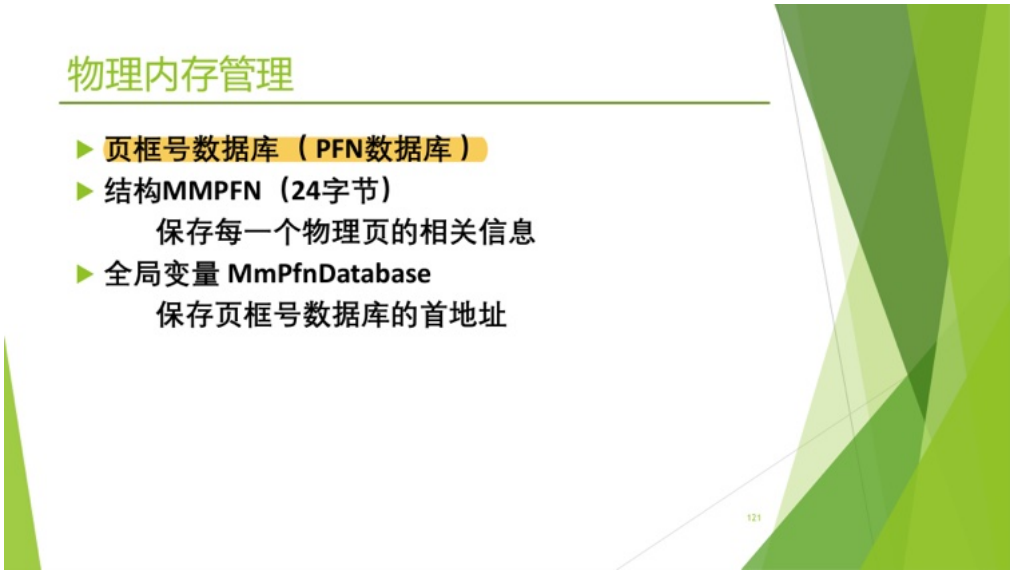
图示：区域对象 + mmap 行为；缺页时自动从映射文件读入。



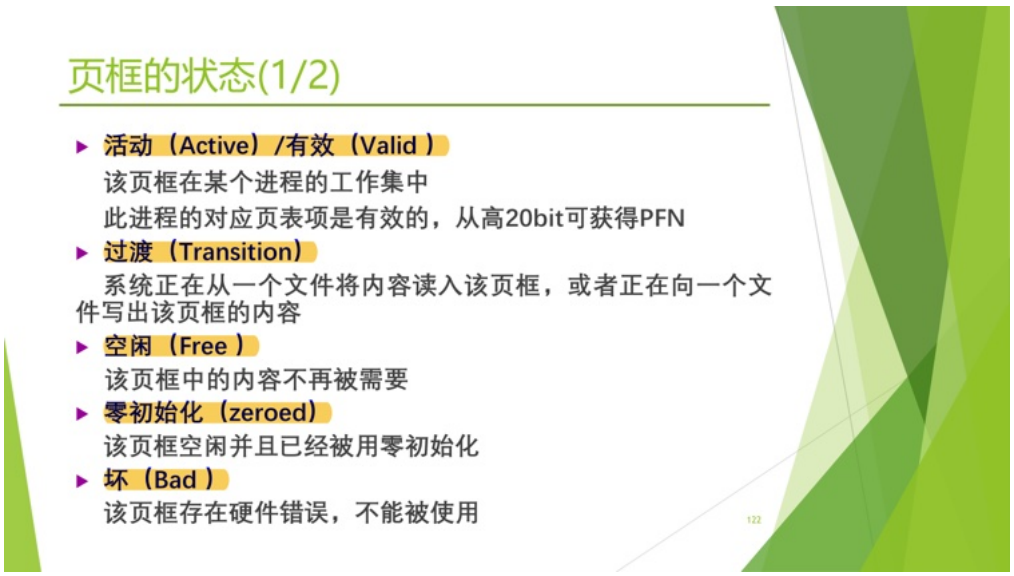
图示：HeapCreate / HeapAlloc / HeapFree / HeapDestroy；适合大量小型内存申请。



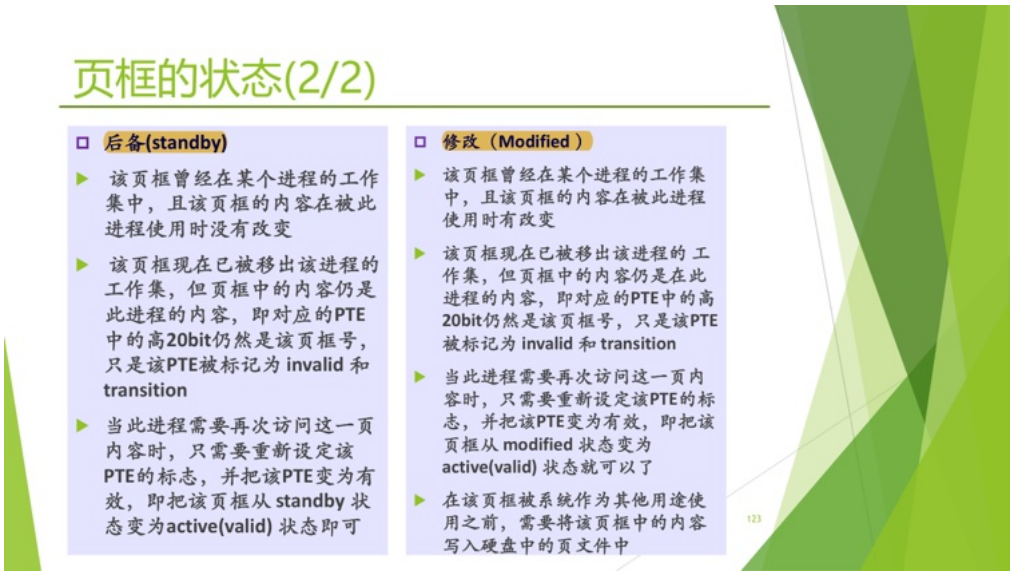
图示：保留若干空闲页框；置换出去的页可被快速恢复。



图示：MMPFN 结构（24 字节）；MmPfnDatabase 全局变量保存数据库首地址。



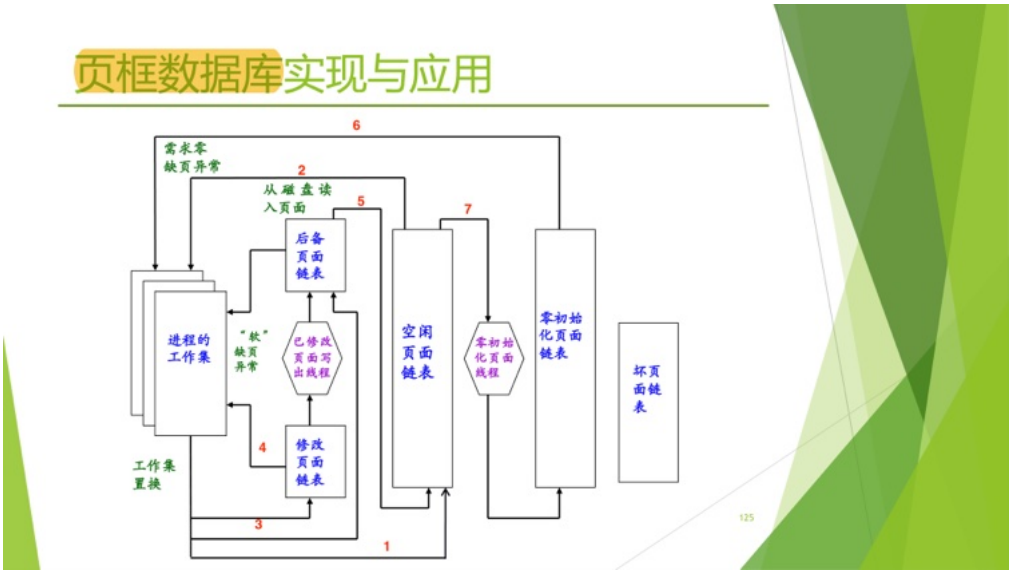
图示：活动 / 过渡 / 空闲 / 零初始化 / 坏 / 修改 / 后备。



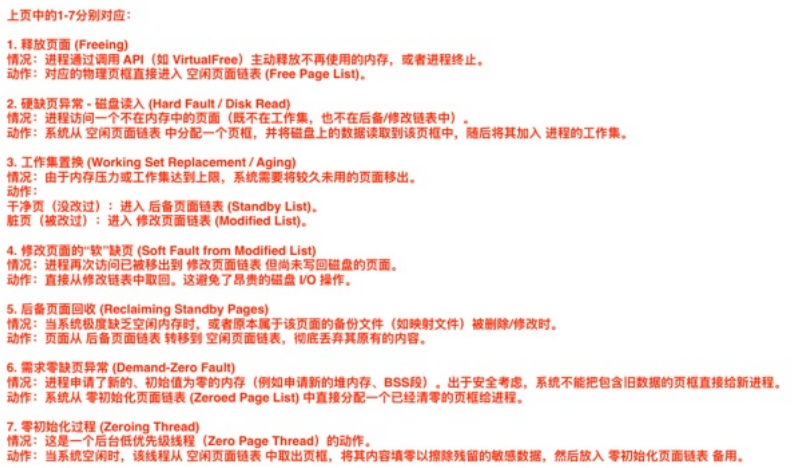
图示：Modified 与 Standby 之间的转换路径，以及如何被快速恢复到 Active。



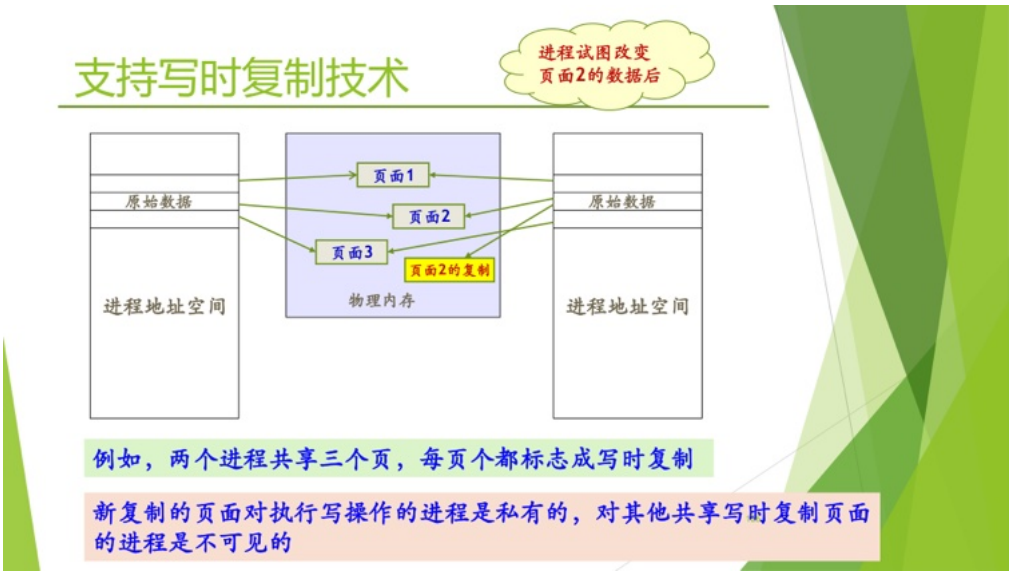
图示：页表中页框号同时挂到 PFN 数据库中的对应项；链表头管理空闲、修改、后备等。



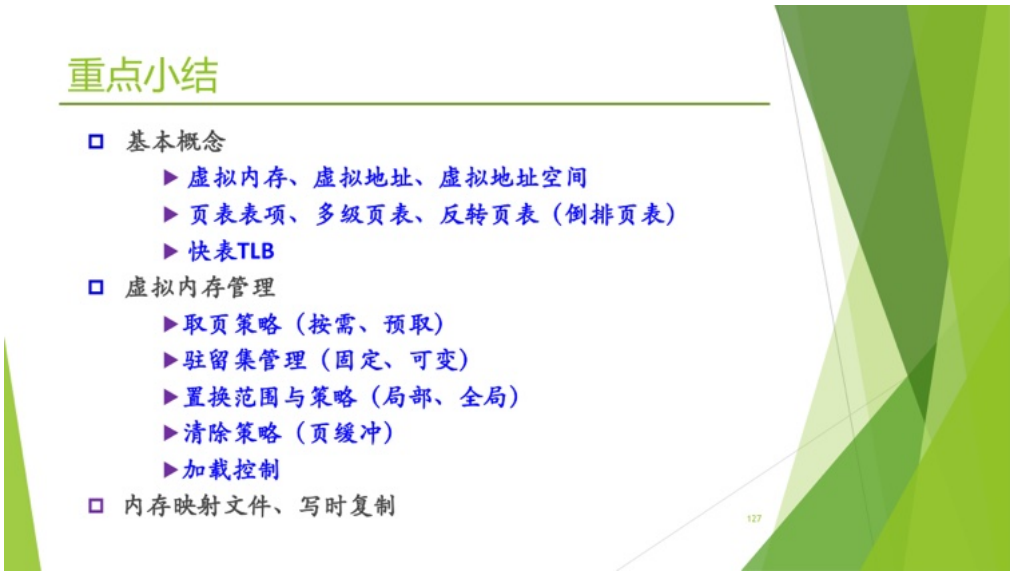
图示：进程工作集、后备链表、修改链表、空闲链表、零初始化链表、坏页链表、零初始化线程、修改页写出线程、需求零缺页异常、软缺页异常之间的循环。



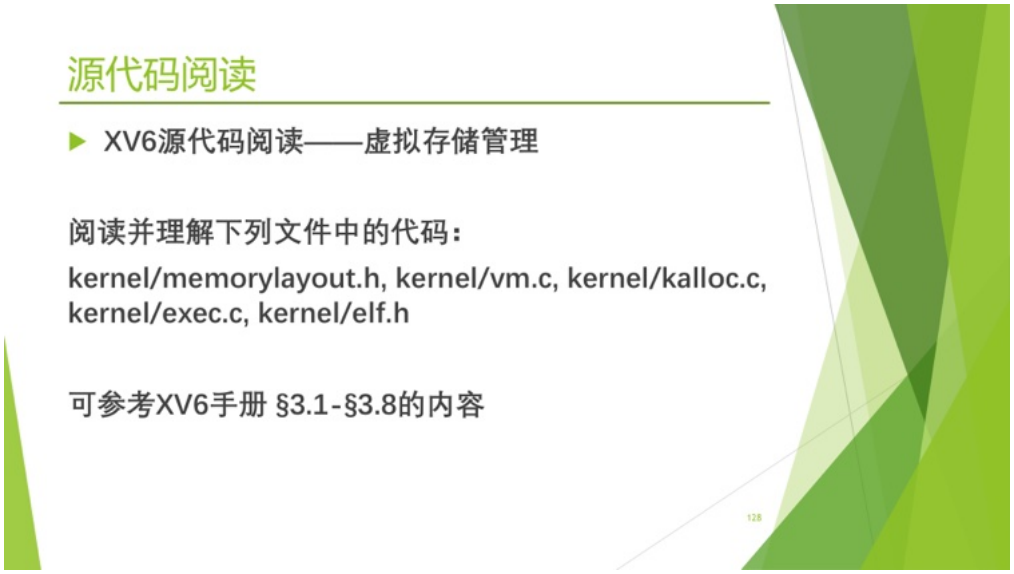
图示：1–7 步详解：从释放、磁盘读入、工作集置换、需求 0、软缺页等。



图示：两个进程共享三个页面；都标记为 COW；当其中一个进程写时，复制对应页。



图示：基本概念、PTE / 多级页表 / 反置页表 / TLB、虚拟内存管理、各种策略、内存映射文件、COW。



图示：阅读 `kernel/memlayout.h`、`vm.c`、`kalloc.c`、`exec.c`、`elf.h`，对应 xv6-book 第 3.1–3.8 节。

LECTURE 7

第 7 讲

文件系统 1

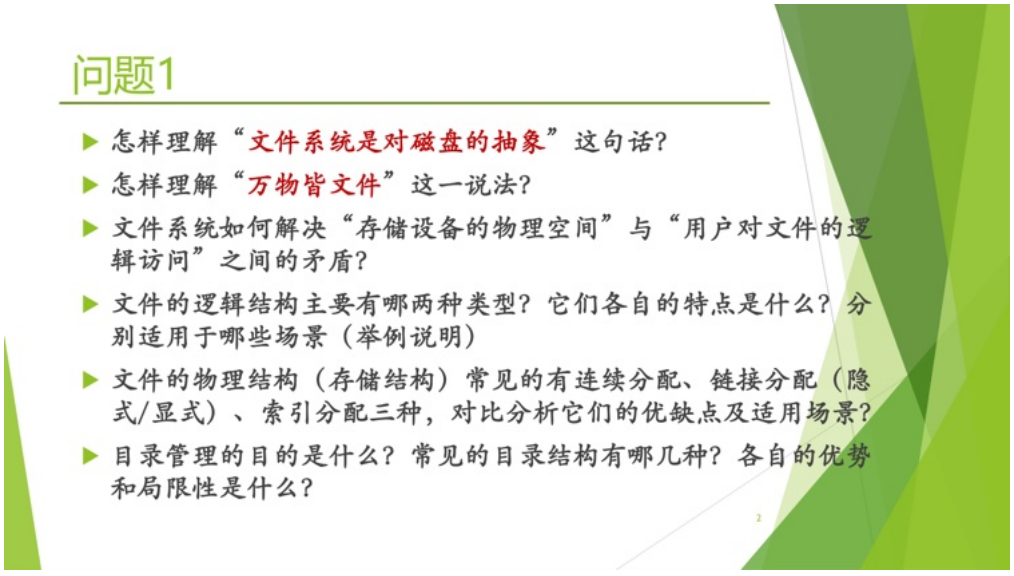
Operating Systems Course Notes

7.1 本讲导读

- 这一讲回答“文件系统如何把磁盘抽象成文件和目录”。
- 学习主线：
  1. 文件、目录、文件系统的基本概念；
  2. 文件控制块、目录项、路径名解析；
  3. 文件的逻辑结构与物理结构；
  4. 文件系统在磁盘和内存中的布局；
  5. 磁盘空间管理、文件操作与文件共享；
  6. FAT、UNIX、NTFS 与 XV6 文件系统实例。

> **关键提醒：**文件系统的本质是“名字空间到磁盘空间的映射”。用户看到的是文件名、目录、权限和读写接口；内核真正管理的是块号、inode、缓存、空闲空间和一致性。

7.2 开篇问题

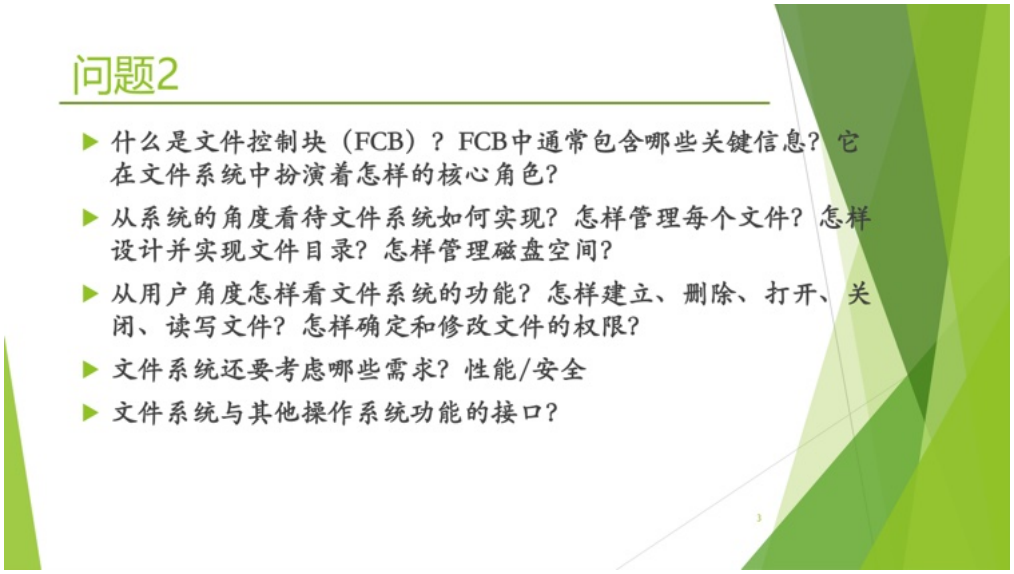


图示：文件系统作为磁盘抽象、万物皆文件、逻辑访问与物理空间映射、文件结构与目录结构。

> **思考题：**怎样理解“文件系统是对磁盘的抽象”?

解答：磁盘硬件只提供“读写某个扇区/块”的低级接口，用户并不想关心柱面、磁头、扇区、LBA、坏块和寻道。文件系统把这些物理细节封装起来，提供“按文件名打开、按偏移读写、按目录组织、按权限保护”的逻辑接口。因此，文件系统把一维或多维的磁盘块空间抽象成了带名字、层次、权限和类型的文件空间。

> **思考题：**怎样理解“万物皆文件”?



图示：FCB、目录实现、磁盘空间管理、用户视角功能、性能与安全需求。

解答： UNIX/Linux 把普通文件、目录、设备、管道、套接字、伪文件系统等统一纳入文件接口。这样应用程序可以用 `open/read/write/close/ioctl` 等相对统一的系统调用访问不同资源。它不是说所有资源在物理上都是磁盘文件，而是说内核向用户态暴露了统一的“文件对象 + 操作表”抽象。

> **思考题：** 文件系统如何解决“存储设备物理空间”和“用户逻辑访问”之间的矛盾？

解答： 文件系统通过多层映射完成转换：

- 目录把路径名映射到文件元数据对象，例如 inode 或 FCB；
- 文件元数据把逻辑块号映射到物理块号；
- 块设备层把物理块号转换为设备命令；
- 缓冲缓存、预读、延迟写等机制隐藏低速磁盘访问；
- 权限、链接计数、日志等机制保证共享、保护和一致性。

> **思考题：** 文件的逻辑结构主要有哪两类？适用场景是什么？

解答：

- **流式文件（Stream File）**：文件是字节序列，应用自己解释内容；UNIX 普通文件就是典型例子，适合文本、二进制程序、图片等通用数据。
- **记录式文件（Record File）**：文件由记录组成，每条记录可以定长或变长；适合早期事务系统、数据库风格数据、批处理记录文件。

现代通用 OS 通常提供字节流抽象，把更复杂的记录、索引、B+ 树结构交给数据库或应用层实现。

> **思考题：**连续分配、链接分配、索引分配各有什么优缺点？

解答：

物理结构	记录方式	优点	缺点	适用场景
连续分配	起始块 + 长度	顺序/随机访问都快，实现简单	外部碎片，文件增长困难	只读介质、大文件预分配
隐式链接	每块存下一块指针	无外部碎片，易扩展	随机访问慢，指针损坏影响大	早期简单文件系统
显式链接 FAT	FAT 表存链	块内不存指针，便于扫描链	FAT 大时占内存，随机访问仍需追链	FAT12/16/32
索引分配	inode/索引块存块号	随机访问好，支持稀疏和大文件	小文件索引开销，超大文件需多级索引	UNIX/ext/xv6

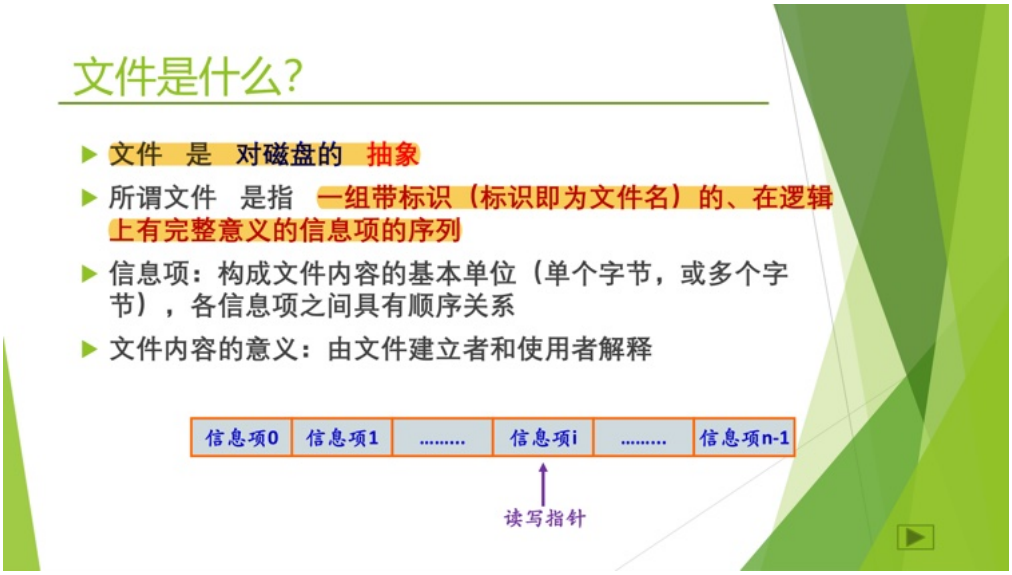
> **思考题：**目录管理的目的是什么？常见目录结构有哪些？

解答：目录的目标是按名存取、组织文件、支持共享与保护。常见结构包括：

- 单级目录：实现简单，但所有文件同名冲突严重；
- 二级目录：每个用户一个目录，缓解冲突；
- 树形目录：支持层次化组织，是现代 OS 主流；
- 无环图目录：通过链接支持共享，但要避免环；
- 一般图目录：表达力强，但垃圾回收、遍历和权限复杂。

7.3 文件与文件目录

7.3.1 文件的基本概念



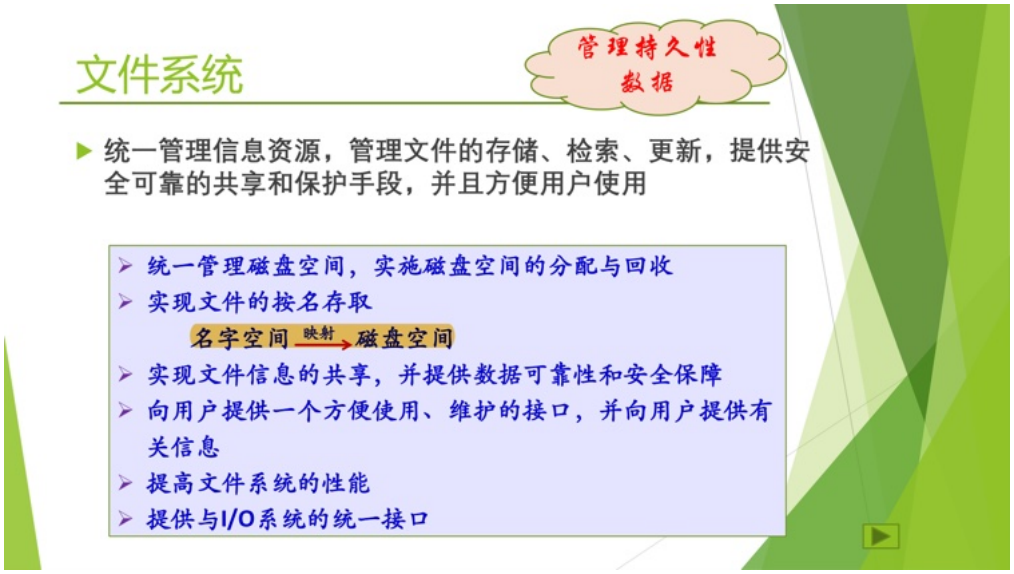
图示：文件是带标识的信息项序列，文件系统隐藏磁盘物理细节。

- **文件（File）**是具有名字的一组相关信息项的集合。
- 文件内容由创建者和使用者解释，操作系统一般不理解内容语义。
- 文件通常包含：
  - 文件名；
  - 文件类型；
  - 文件大小；
  - 所有者和权限；
  - 时间戳；
  - 存放位置；
  - 读写指针或打开状态。

7.3.2 文件类型

- **普通文件**：文本、二进制、可执行程序、图片等。
- **目录文件**：保存目录项，用于名字到文件元数据的映射。
- **字符设备文件**：键盘、串口、终端等按字节流访问的设备。
- **块设备文件**：磁盘、分区、光驱等按块访问的设备。
- **管道文件**：用于进程间通信。
- **套接字文件**：用于本机或网络通信。
- **符号链接**：保存另一个路径名的特殊文件。

7.3.3 文件控制块 FCB



图示：文件系统通过文件控制块记录文件属性和物理位置。

文件控制块（File Control Block, FCB）是操作系统管理文件的核心元数据结构。它通常保存：

- 文件标识：文件名或内部编号；

- 文件类型：普通、目录、设备等；
- 物理位置：连续分配的起始块，或索引表/inode 指针；
- 文件大小：字节数、块数；
- 保护信息：所有者、组、访问权限、ACL；
- 管理信息：创建时间、修改时间、访问时间；
- 共享信息：链接计数、打开计数；
- 其他控制位：只读、隐藏、系统、归档等。

在 UNIX 风格系统中，目录项通常只保存“文件名 + inode 号”，真正的 FCB 信息放在 inode 中。这种分离让硬链接成为可能。

7.3.4 文件操作

- `create`：创建新文件，本质是分配 FCB/inode 并插入目录项。
- `delete/unlink`：删除目录项，若链接计数归零再释放 inode 和数据块。
- `open`：路径名解析、权限检查、建立打开文件表项。
- `close`：减少打开计数，必要时回写缓存并释放内存结构。
- `read/write`：按当前偏移或指定偏移进行逻辑块到物理块转换。
- `append`：把写入偏移移动到文件末尾。
- `seek`：修改当前文件偏移，实现随机访问。
- `get/set attributes`：读取或修改元数据。
- `rename`：修改目录项名字，通常不移动数据块。

7.4 目录结构与路径名解析

7.4.1 目录文件

目录文件（Directory File）保存目录项集合。目录项可以是：

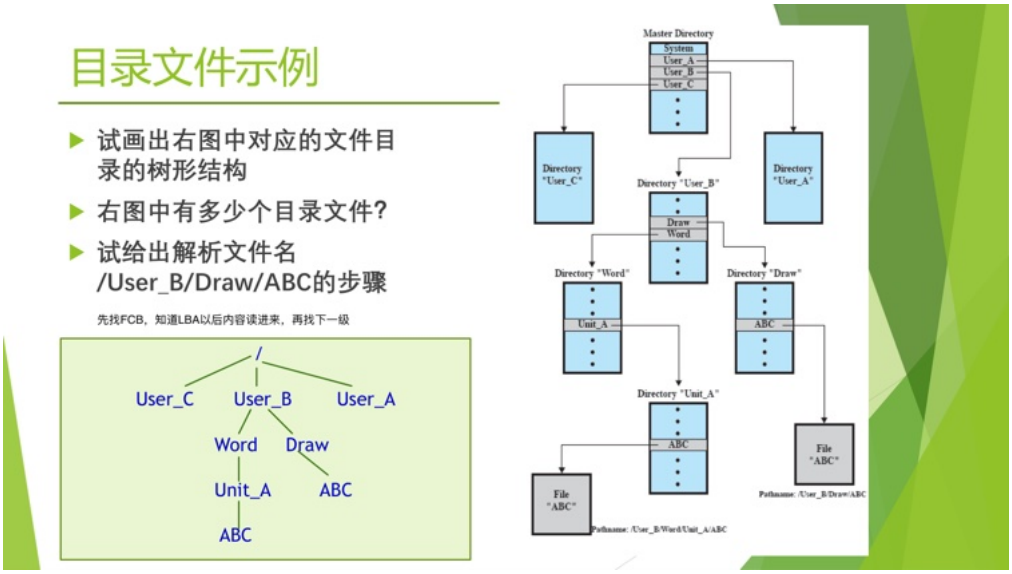
- 简单形式：目录项就是完整 FCB；
- UNIX 形式：目录项只保存文件名和 inode 号；
- 哈希/树索引形式：用于加速大目录查找。

7.4.2 目录结构

> **思考题：**试画出图中对应的树形目录结构，右图中有多少个目录文件？给出解析 `/User_B/Draw/ABC` 的步骤。

解答：

- 树根是 Master Directory / Root。



图示：树形目录结构与路径 `/User_B/Draw/ABC` 的解析。

- 根目录下至少包含 `User_A`、`User_B`、`User_C` 等用户目录。
- `User_B` 下有 `Word`、`Draw` 等子目录。
- `Draw` 目录下有普通文件 `ABC`。
- 目录文件数量等于图中目录节点数量，不包括普通文件节点。

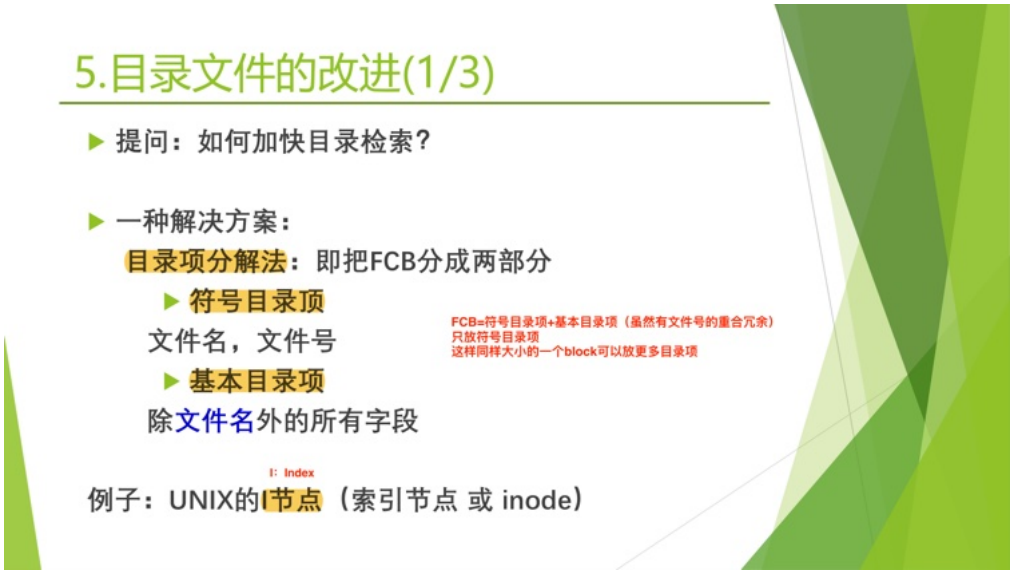
路径解析步骤：

1. 从根目录 `/` 的 inode/FCB 开始；
2. 在根目录文件中查找名字 `User_B`，得到 `User_B` 的目录 inode；
3. 读取 `User_B` 目录文件，查找 `Draw`；
4. 读取 `Draw` 目录文件，查找 `ABC`；
5. 得到 `ABC` 的 inode/FCB；
6. 根据权限、打开模式和文件类型决定是否允许后续 `open/read/write`。

7.4.3 加快目录检索

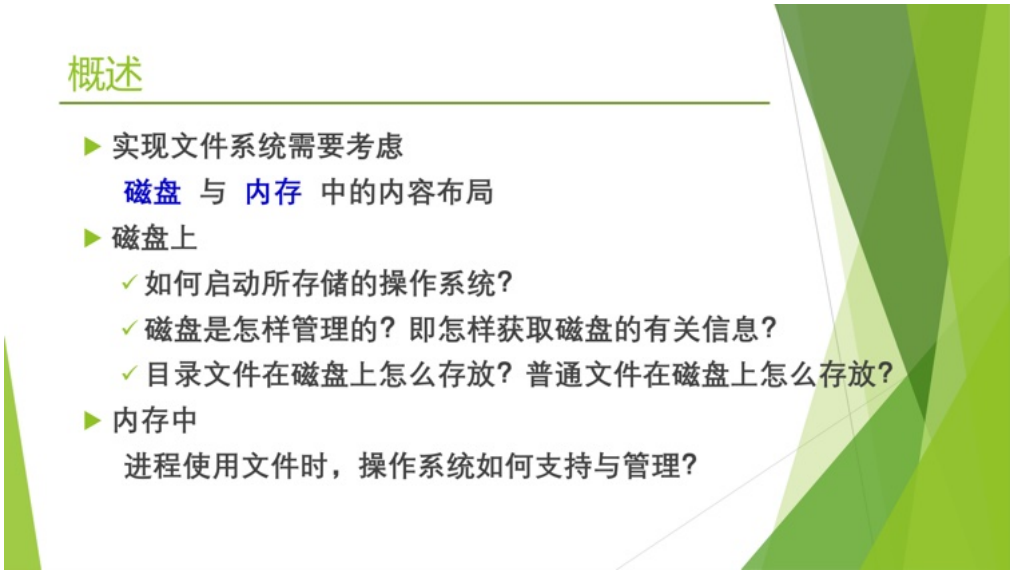
- 线性目录检索简单，但大目录中平均查找代价高。
- 目录项分解法把文件名和元数据分开：
  - 符号目录项保存文件名和指针；
  - 基本目录项保存完整 FCB。
- UNIX 的“目录项 + inode”正是这种思想。
- 现代文件系统还会使用哈希目录、B 树目录或 extent tree 加速查找。

7.5 文件系统实现



图示：把目录项分成符号目录项和基本目录项，减少平均访盘次数。

7.5.1 磁盘与内存中的布局



图示：文件系统实现需要同时设计磁盘布局和内存数据结构。

文件系统实现需要回答两类问题：

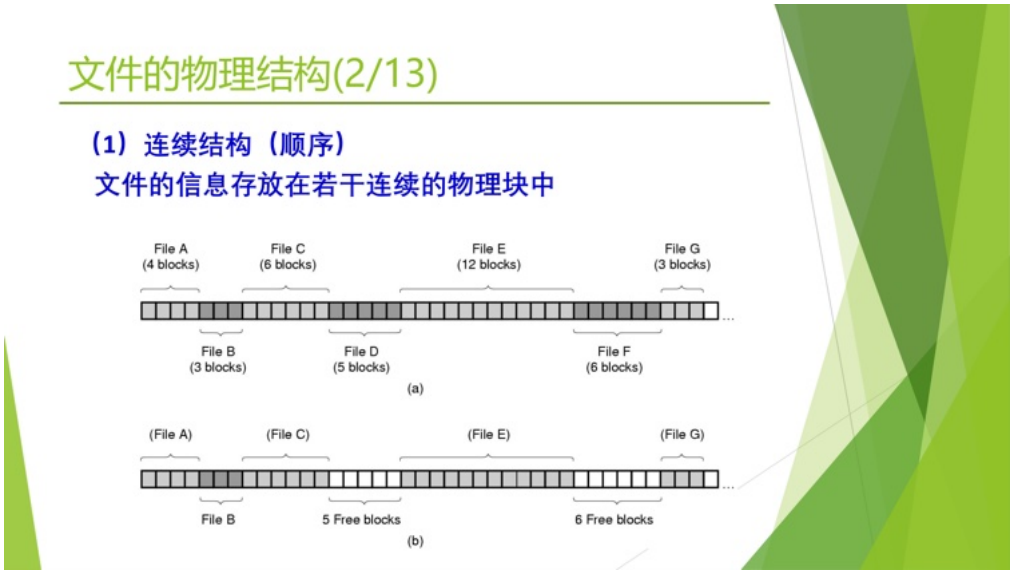
- 磁盘上：
  - 如何启动所存储的操作系统；
  - 如何描述文件系统整体信息；
  - inode/FCB、目录、普通文件如何存放；
  - 空闲块如何管理；
  - 日志或恢复信息如何组织。
- 内存中：
  - 如何缓存超级块、inode、目录项和数据块；

- 如何表示打开文件；
- 如何同步并发访问；
- 如何把系统调用转换为块设备请求。

典型磁盘布局：

```
boot block | super block | log | inode blocks | bitmap/free map | data blocks
```

7.5.2 文件的物理结构



图示：连续分配用起始块和长度描述文件位置。

连续分配

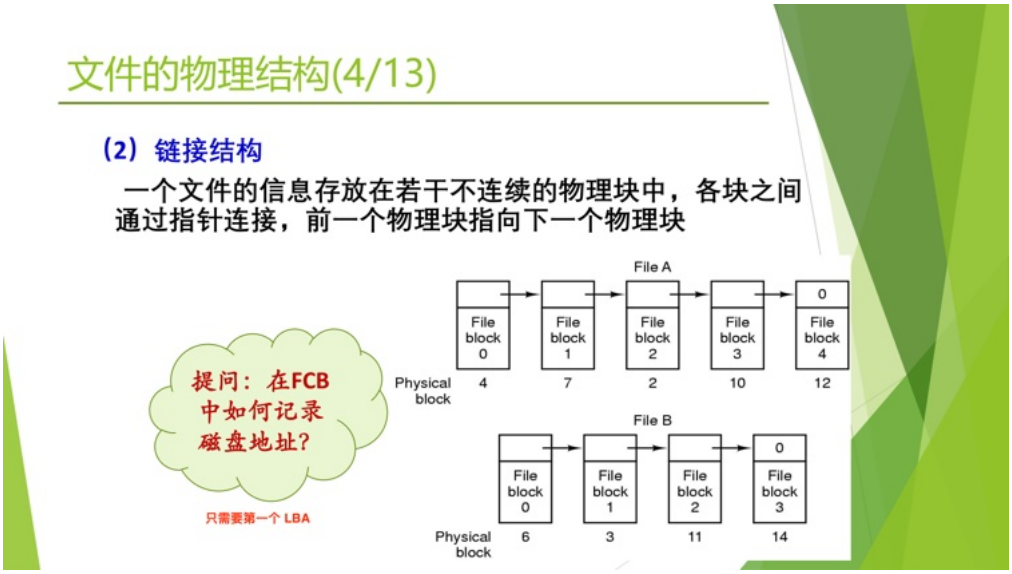
- FCB 只需记录第一个 LBA 和文件长度；
- 顺序读写和随机读写都很快；
- 文件创建时需要估计大小；
- 文件扩展可能需要搬移；
- 容易产生外部碎片。

> **思考题：** 连续分配中 FCB 如何记录磁盘地址？

解答： 记录文件第一个物理块号（或 LBA）以及文件长度/块数。逻辑块号 `i` 对应物理块号 `start + i`。

链接分配

- 隐式链接：每个数据块保存下一块块号。
- 显式链接：把所有下一块指针集中存到 FAT 表。

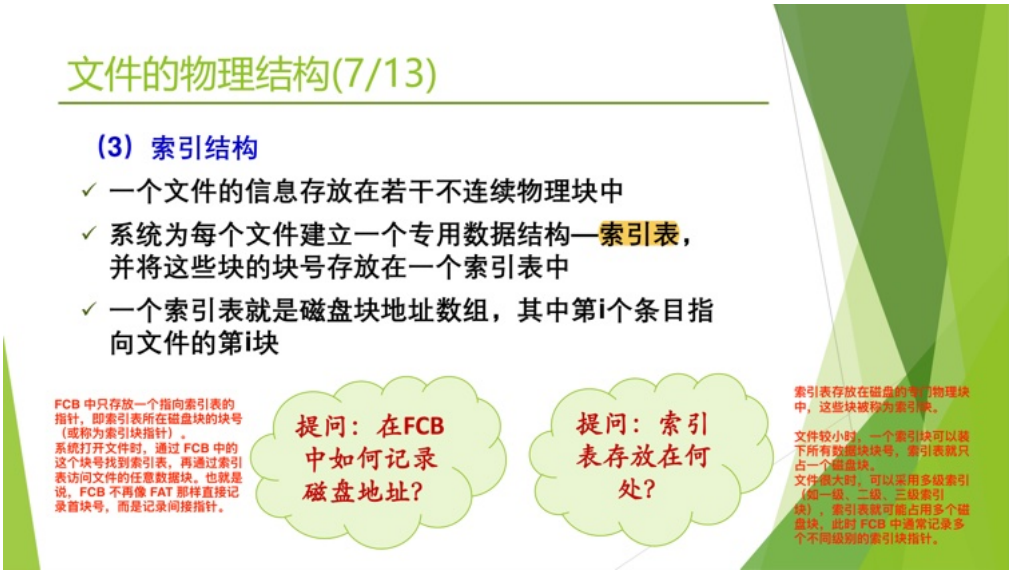


图示：链接分配通过链指针或 FAT 表记录文件块序列。

- FCB 只需保存起始块号。
- 优点是无外部碎片、文件扩展方便。
- 缺点是随机访问慢，链表损坏会影响文件完整性。

> **思考题：**某文件的起始块号从何处得到?

解答：从该文件的 FCB/目录项中得到。FAT 表记录的是“当前块的下一块”，不是文件入口；文件入口必须由目录项或 FCB 给出。



图示：索引分配把逻辑块号映射集中放到索引块或 inode 中。

索引分配

- FCB 中保存索引表所在块号；
- 索引表保存文件各逻辑块对应的物理块号；

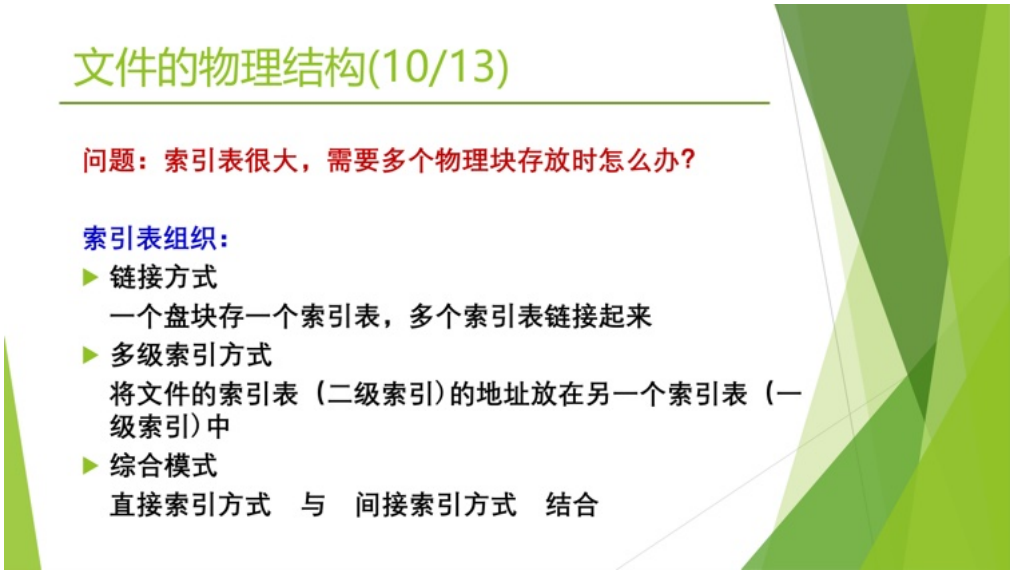
- 随机访问效率高；
- 小文件可能浪费索引块；
- 大文件需要链接索引、多级索引或混合索引。

> **思考题：**索引表很大，需要多个物理块存放时怎么办？

解答：

- 链接索引：多个索引块串成链，适合顺序访问；
- 多级索引：一级索引指向二级索引块，二级索引再指向数据块；
- 混合索引：直接索引 + 一级间接 + 二级间接 + 三级间接，兼顾小文件和大文件。

7.5.3 UNIX 多级索引



图示：UNIX inode 使用直接索引、一级间接索引、二级间接索引和三级间接索引。

> **思考题：** UNIX 中采用三级索引结构后，文件最大可达到多少个物理块？假设块大小 512 字节，一个一级索引表可存 256 个物理块号。

解答：若 inode 有 10 个直接块、1 个一级间接、1 个二级间接、1 个三级间接，则最大块数为：

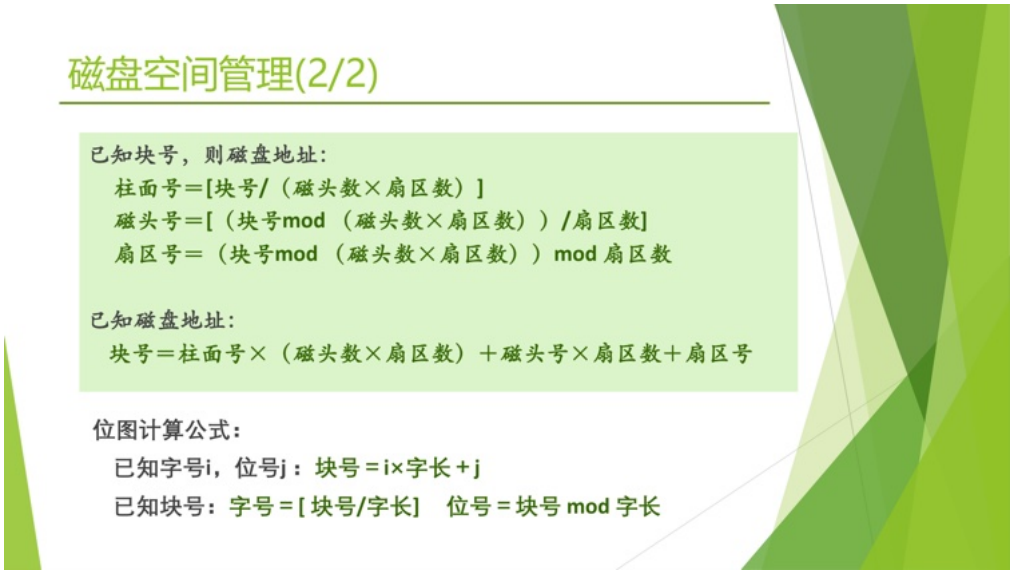
$$10 + 256 + 256^2 + 256^3$$

其中：

- 10 个直接块服务小文件；
- 一级间接可索引 256 个数据块；
- 二级间接可索引 256^2 个数据块；
- 三级间接可索引 256^3 个数据块。

这说明多级索引用很少的 inode 字段同时支持小文件低开销和大文件高容量。

7.5.4 磁盘空间管理



图示：空闲表、空闲链表、位示图和成组链接等空闲块管理方法。

常用方法：

- **空闲表**：记录连续空闲区的起始块和长度，适合连续分配。
- **空闲链表**：把空闲块链接起来，实现简单但遍历慢。
- **位示图（Bitmap）**：每个块一位，`0/1` 表示空闲/占用；适合快速查找连续空间。
- **成组链接**：UNIX 早期方法，把一组空闲块号存入专用块，兼顾空间和效率。

> **思考题：**专用块的位置在哪里？

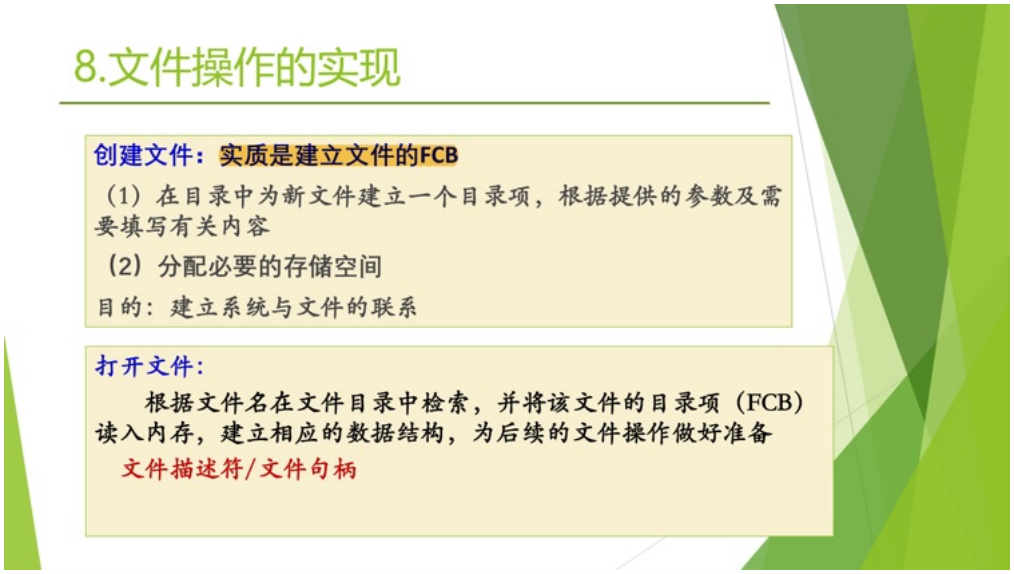
解答：专用块既可能在磁盘上持久化保存，也会在内存中保留当前可用的一组空闲块号。磁盘上的专用块保证重启后仍可恢复空闲空间信息；内存中的副本提高分配和回收效率。

7.6 文件操作实现与共享

7.6.1 创建、打开、读写

创建文件：

1. 检查路径和权限；
2. 在目标目录中申请空目录项；
3. 分配 inode/FCB；
4. 必要时为初始内容分配数据块；
5. 写回目录和元数据。



图示：创建、打开、读、写等文件操作最终都要落到目录、FCB/inode 和磁盘块分配。

打开文件：

1. 路径名解析得到 inode；
2. 检查权限和打开标志；
3. 在系统打开文件表建立表项；
4. 在进程文件描述符表中分配 fd；
5. 返回 fd 给用户进程。

读文件：

1. 检查 fd 是否有效、是否可读；
2. 根据当前偏移计算逻辑块号；
3. 通过 inode/索引结构转换为物理块号；
4. 读缓冲缓存或发起磁盘 I/O；
5. 拷贝到用户缓冲区；
6. 更新文件偏移和访问时间。

> **思考题：** 文件尚未打开时能否直接读?

解答： 通常不能。`read` 接收的是文件描述符，文件描述符来自 `open`。没有 `open` 就没有进程打开文件表项、当前偏移、访问权限和引用计数，内核无法安全地执行读操作。

7.6.2 rename 与 copy

> **思考题：** 怎样实现系统调用 `rename`?

解答： 同一文件系统内的 `rename` 通常只修改目录项：



图示：rename 和 copy 的实现讨论。

1. 解析旧路径，得到旧目录项和 inode；
2. 解析新路径所在目录，检查权限与重名处理；
3. 在新目录中加入或替换目录项；
4. 删除旧目录项；
5. 必要时更新目录时间戳和链接计数；
6. 使用日志或事务保证“要么旧名存在，要么新名存在”，避免崩溃后两边都丢。

> **思考题：** 怎样实现系统调用 `copy`?

解答： `copy` 不是传统 UNIX 内核最小接口，通常由用户态程序实现：

1. `open(src, O_RDONLY)`；
2. `open(dst, O_CREAT|O_WRONLY|O_TRUNC)`；
3. 循环 `read` 源文件并 `write` 到目标文件；
4. 复制权限、时间戳和扩展属性；
5. `close` 两个文件。

现代文件系统可优化为 reflink/COW copy，即先共享物理块，写入时再复制。

7.6.3 硬链接与软链接

- **硬链接**：多个目录项指向同一个 inode；删除一个名字只是减少链接计数。
- **软链接**：一个特殊文件保存目标路径；可跨文件系统，目标不存在时成为悬空链接。

> **思考题：** 硬链接怎样删除文件?

解答： 删除目录项时 inode 链接计数减一。只有当链接计数为 0 且没有打开文件引用时，内核才

释放 inode 和数据块。

> **思考题：**软链接有什么问题?

解答：软链接需要额外路径解析，开销较高；可以形成环；目标删除后会悬空；跨网络或挂载点时还要处理权限、路径语义和命名空间变化。

7.7 文件系统实例

7.7.1 FAT

- FAT 表保存“当前块到下一块”的映射。
- 目录项保存文件起始簇号。
- 优点是结构简单、兼容性好。
- 缺点是大卷上 FAT 表庞大，可靠性和权限机制较弱。
- xv6-k210 实验中使用 `mkfs.vfat -F 32` 创建 FAT32 文件系统，说明 FAT 仍适合简单可移动介质和教学场景。

7.7.2 UNIX 文件系统

- inode 保存文件类型、权限、大小、链接计数和块索引。
- 目录项保存文件名和 inode 号。
- 打开文件表分离“文件对象”和“进程 fd”。
- 多级索引支持大小文件。
- 缓冲缓存提高块访问效率。

7.7.3 NTFS

- NTFS 的核心是 **主文件表（Master File Table, MFT）**。
- 每个文件和目录都由 MFT 文件记录描述。
- 小属性可直接驻留在 MFT 记录中。
- 大文件通过运行列表描述磁盘簇范围。
- NTFS 支持日志、ACL、压缩、加密、稀疏文件等机制。

7.8 XV6 文件系统代码阅读

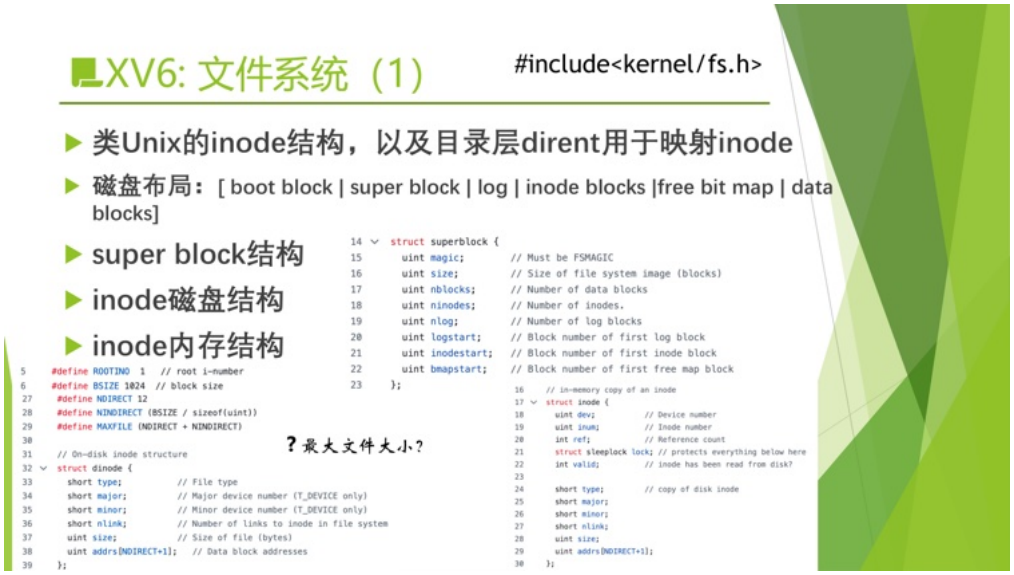
> **代码阅读题：** XV6 文件系统的磁盘布局是什么?

解答： XV6 的磁盘布局为:

```
[ boot block | super block | log | inode blocks | free bit map | data blocks ]
```



图示：NTFS 以 MFT 作为卷结构核心，用文件记录数组描述文件和目录。

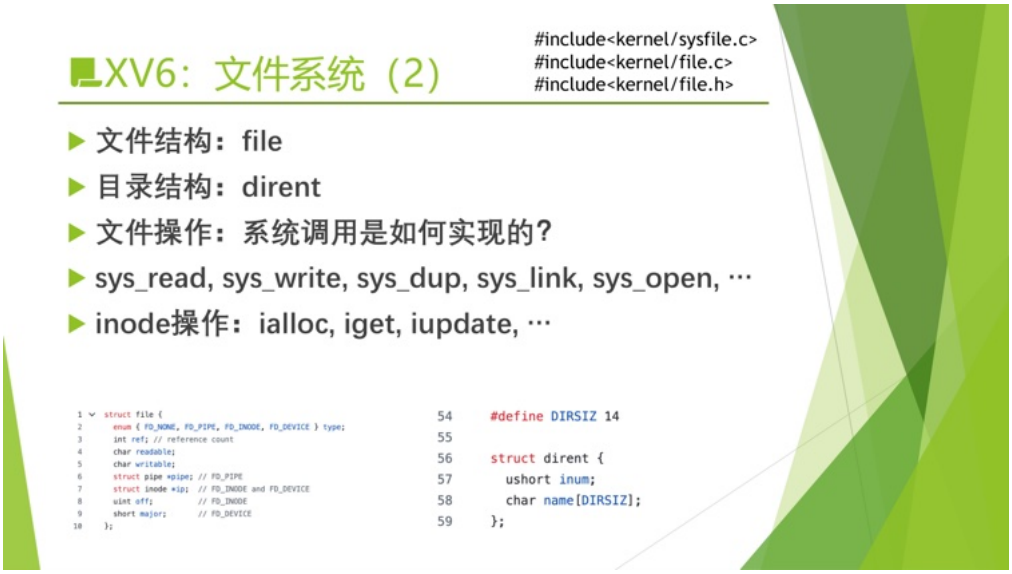


图示：XV6 磁盘布局、superblock、inode、dirent 与最大文件大小问题。

- `boot block`：启动相关；
- `super block`：记录文件系统规模、inode 数量、日志区位置等；
- `log`：提供简单事务日志；
- `inode blocks`：保存磁盘 inode；
- `free bitmap`：记录空闲数据块；
- `data blocks`：保存普通文件和目录内容。

> **代码阅读题：** XV6 最大文件大小如何计算?

解答： XV6 inode 中 `addrs` 包含若干直接块和一个一级间接块。若直接块数为 `NDIRECT`，间接块



图示：XV6 的 `file`、`dirent`、`sys_read`、`sys_write`、`sys_open` 和 inode 操作。



图示：XV6 文件系统调用最终落到 inode、目录项和块读写。

可容纳 `NINDIRECT = BSIZE / sizeof(uint)` 个块号，则最大文件块数是：

$$NDIRECT + NINDIRECT$$

最大字节数是：

$$(NDIRECT + NINDIRECT) \times BSIZE$$

这比 UNIX 三级索引简单，适合教学操作系统，但不能支持非常大的文件。

> **代码阅读题：** `sys_read` / `sys_write` 如何实现？

解答：

1. 系统调用层从用户参数中取出 fd、缓冲区地址和字节数；
2. 根据 fd 找到进程打开文件表中的 `struct file`；
3. 检查文件是否可读/可写；
4. 如果是 inode 文件，调用 `readi` / `writei`；
5. `readi` / `writei` 通过 `bmap` 把文件逻辑块号映射到磁盘块号；
6. 通过 buffer cache 读写块；
7. 更新文件偏移、inode 大小和日志事务。

> **代码阅读题：** `sys_open` 的关键步骤是什么？

解答：

1. 解析路径名；
2. 如果带 `O_CREATE`，创建 inode；
3. 否则查找已有 inode；
4. 检查文件类型和访问模式；
5. 分配 `struct file`；
6. 分配 fd；
7. 设置 `readable/writable/off/ip` 等字段；
8. 返回 fd。

7.9 本讲小结

- 文件系统用文件和目录抽象磁盘块。
- FCB/inode 是文件元数据核心，目录负责名字映射。
- 连续、链接、索引分配分别在性能、空间利用和扩展性上取舍。
- UNIX inode 的混合索引结构是“照顾小文件、兼顾大文件”的经典设计。
- 文件操作本质上是路径解析、权限检查、逻辑块到物理块转换、缓存和元数据更新。
- XV6 提供了简洁的类 UNIX 文件系统实现，是理解真实文件系统的最小模型。

LECTURE 8

第 8 讲

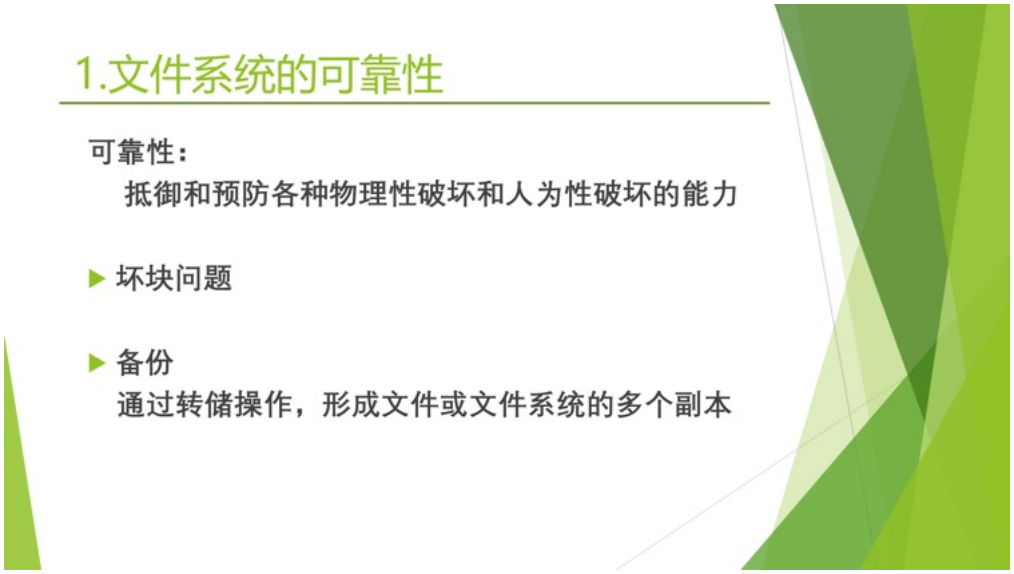
文件系统 2

Operating Systems Course Notes

8.1 本讲导读

- 本讲延续第 7 讲，从“文件系统怎么组织文件”进入“文件系统如何可靠、快速、安全地运行”。
- 知识脉络：
  1. 文件系统可靠性：坏块、备份、恢复；
  2. 文件系统一致性：`fsck`、写入策略、事务日志；
  3. 文件系统安全性：身份认证、ACL、Capability、UNIX 权限；
  4. 文件系统性能：缓存、预读、磁盘调度、RAID；
  5. 文件系统结构：分层结构、VFS、LFS、日志文件系统；
  6. 分布式文件系统：Ceph、HDFS 等；
  7. XV6 buffer cache 与日志系统。

8.2 文件系统可靠性



图示：文件系统可靠性包括坏块问题和备份，备份可分全量、增量、物理、逻辑。

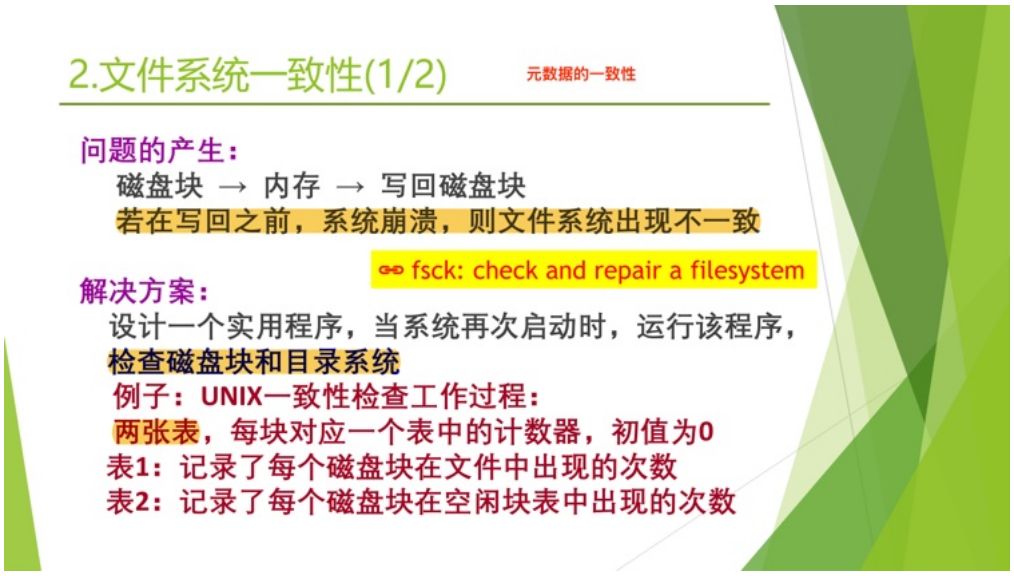
8.2.1 坏块问题

- 磁盘块可能因物理损坏无法正确读写。
- 文件系统需要识别、隔离坏块，避免把新数据分配到坏块上。
- 现代磁盘控制器也会做坏块重映射，但文件系统仍要处理 I/O 错误。

8.2.2 备份方式

方式	思想	优点	缺点
全量转储	定期复制全部文件	恢复简单	时间和空间开销大
增量转储	只复制上次以来修改的文件	开销小	恢复需要多份备份
物理转储	按磁盘块顺序复制	快，保留底层状态	不理解文件语义
逻辑转储	从目录树递归复制文件	可恢复单个文件/目录	需要遍历元数据

8.3 文件系统一致性



图示：磁盘块先进入内存缓存，再写回磁盘；崩溃可能导致文件系统不一致。

8.3.1 为什么会不一致

一次文件系统操作通常要修改多个位置：

- 数据块；
- inode 或 FCB；
- 目录项；
- 空闲块位图；
- 日志或超级块统计信息。

如果系统在这些修改全部写回前崩溃，磁盘可能停在中间状态。例如：一个块已经写入文件 inode，但空闲位图仍显示空闲；或目录项已经创建，但 inode 尚未初始化。

8.3.2 `fsck` 检查

`fsck` 的基本思想是重新扫描磁盘元数据，构造一致性视图：

1. 遍历所有 inode 和间接块，统计每个块被文件引用的次数；

2. 扫描空闲块表或位图，统计每个块在空闲结构中出现的次数；
3. 对比两张表：
   - 文件引用 1 次、空闲 0 次：正常已分配块；
   - 文件引用 0 次、空闲 1 次：正常空闲块；
   - 文件引用 0 次、空闲 0 次：块泄漏，应加入空闲表；
   - 文件引用大于 0、空闲大于 0：已分配块误在空闲表中，应从空闲表删除；
   - 文件引用大于 1：多个文件共享同一块，通常复制该块并修复引用。

8.3.3 写入策略



图示：通写、延迟写和可恢复写在性能与一致性之间取舍。

- **通写（write-through）**：内存修改立即写磁盘，一致性好，性能差。
- **延迟写（lazy-write/write-back）**：先写缓存，稍后批量回写，性能好，崩溃风险高。
- **可恢复写（transaction log/journaling）**：先写日志，再写真实位置，崩溃后根据日志重做或撤销。

> **思考题：**为什么事务日志可以兼顾安全性和性能？

解答：日志把多个元数据更新包装成事务。提交前，恢复程序认为事务未完成；提交后，即使真实位置还没完全写入，也可以根据日志重做。这样文件系统不必每一步都同步写回真实位置，又能在崩溃后恢复到一致状态。

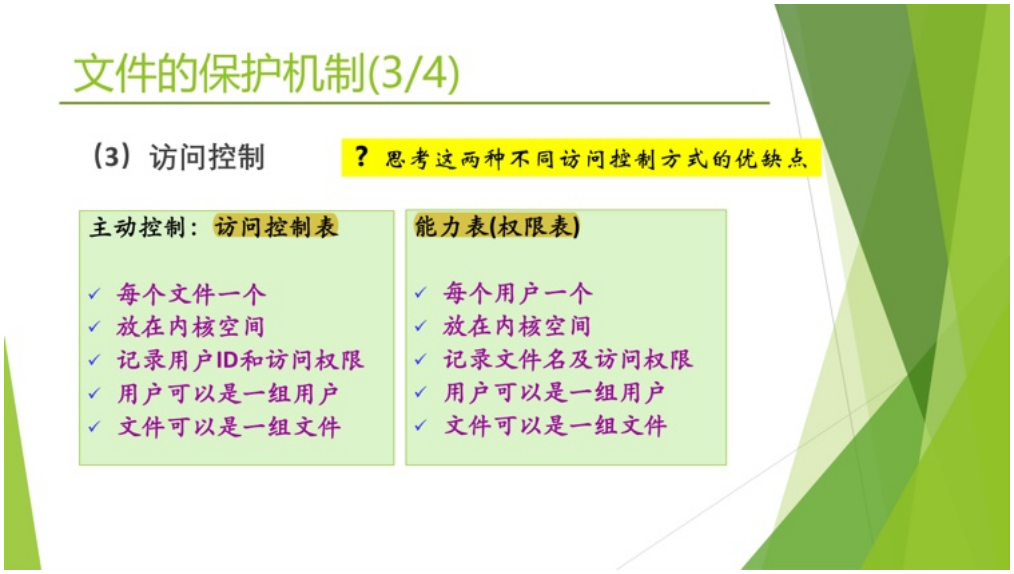
8.4 文件系统安全性

8.4.1 身份认证

- 用户知道什么：口令、密码；

- 用户拥有什么：磁卡、智能卡、USB Key；
- 用户是什么：指纹、虹膜、人脸、声纹、步态；
- 用户能否通过挑战：CAPTCHA、多因素认证。

8.4.2 ACL 与 Capability



图示：ACL 以资源为中心，Capability 以主体为中心。

> **思考题：**访问控制表和能力表的优缺点是什么？

解答：

机制	中心	优点	缺点
ACL	文件/资源	容易回答“谁能访问我”，撤销某个文件上的权限方便	查询某用户全部权限要遍历资源
Capability	用户/主体	容易回答“我能访问谁”，访问时出示能力凭证即可	能力分发后撤销困难，需要失效机制

8.4.3 RBAC 与 ABAC

- **RBAC（Role-Based Access Control）**：用户绑定角色，角色绑定权限。适合组织结构清晰、职责稳定的场景。
- **ABAC（Attribute-Based Access Control）**：根据用户属性、资源属性、环境属性决策。适合复杂细粒度访问控制，例如“医生在工作时间从医院内网访问自己病人的病历”。

8.4.4 UNIX 权限

- 访问者分三类：owner、group、other。
- 操作分三类：read、write、execute。
- 典型权限如 `rwxr-xr--`。



图示：UNIX 使用 owner/group/other 与 read/write/execute 组成 9-bit 权限。

- `chmod 755 file` 表示 owner 可读写执行，group/other 可读可执行。

8.5 数据恢复

> **思考题：**哪些情况下数据不能被恢复？

解答：数据恢复依赖“数据内容尚未被覆盖”。不能恢复的典型情况包括：

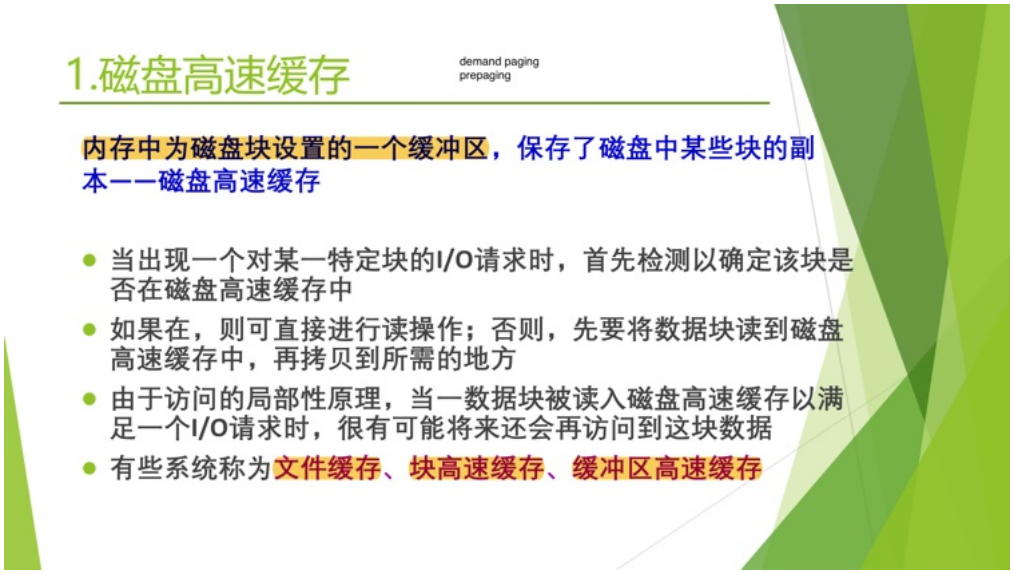
- 原数据块已被新数据覆盖；
- SSD 触发 TRIM，底层块被擦除；
- 加密文件丢失密钥；
- 磁盘发生严重物理损伤；
- 文件系统元数据和数据区均被破坏且无备份。

8.6 文件系统性能

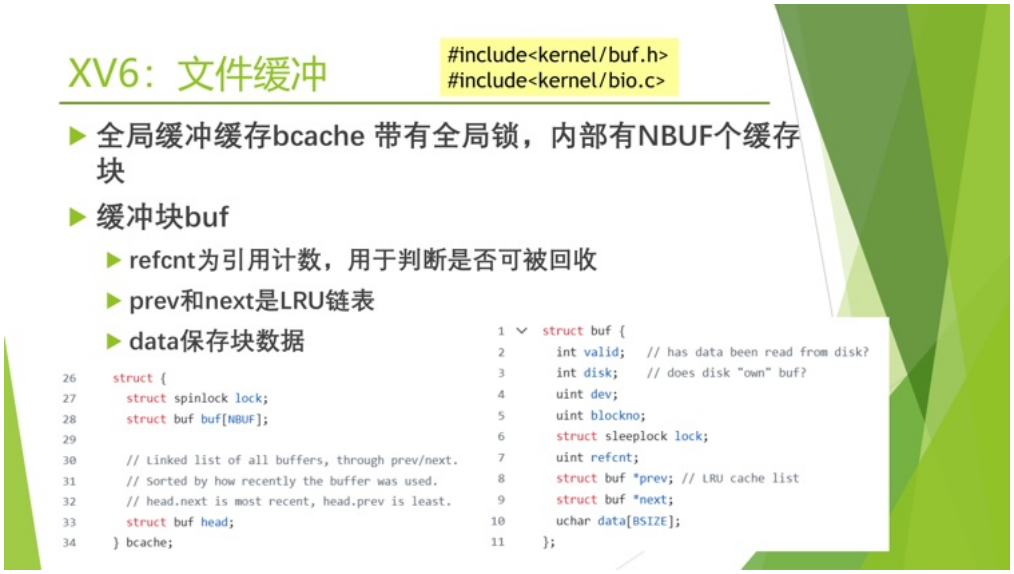
8.6.1 块高速缓存

- 缓存磁盘块副本，减少真实磁盘访问。
- 读命中时直接返回缓存块。
- 读未命中时读取磁盘并放入缓存。
- 写入可采用通写或延迟写。
- 置换算法常用 LRU 或近似 LRU。

> **代码阅读题：** XV6 buffer cache 的作用是什么？



图示：磁盘服务速度和可靠性是文件系统性能瓶颈，应尽量减少磁盘访问。

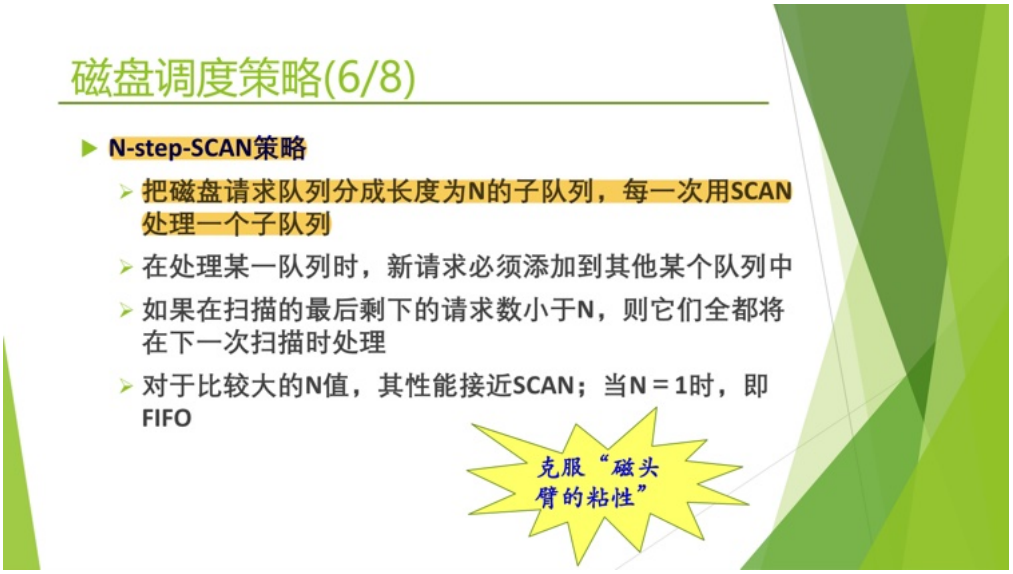


图示：XV6 bcache 由全局锁保护，维护若干缓存块。

解答： XV6 的 bcache 缓存磁盘块，统一服务文件系统和日志系统。每个 buf 保存设备号、块号、有效位、引用计数、睡眠锁和数据。bread 获取块，bwrite 写回块，brelse 释放引用并把最近使用的块移动到 LRU 链表头部。

8.6.2 Windows 文件缓存

- 用户读写通常先访问文件缓存。
- Cache Manager 负责预取、回写、缓存置换。
- FlushFileBuffers 可强制刷新，提升持久性但降低性能。
- 异步 I/O 让线程不必阻塞等待磁盘。

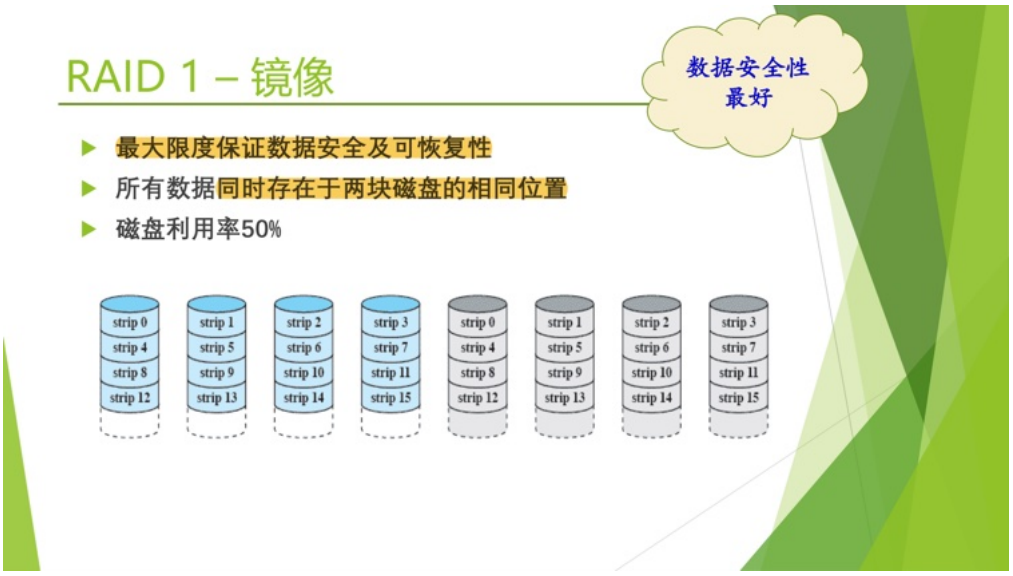


图示：磁盘调度需要降低寻道距离，同时避免饥饿。

8.6.3 磁盘调度

算法	思想	优点	缺点
FCFS	按请求到达顺序服务	公平，实现简单	平均寻道距离可能很大
SSTF	优先服务最近磁道	平均寻道较短	远处请求可能饥饿
SCAN	磁头像电梯一样往返扫描	兼顾效率和公平	两端等待特性不同
C-SCAN	单向扫描，到头后回到起点	等待时间更均匀	回扫成本较高
FSCAN	用两个队列分批处理	避免请求不断插队	实现更复杂

8.6.4 RAID



图示：RAID 通过分条、镜像和校验在性能、容量和可靠性之间取舍。

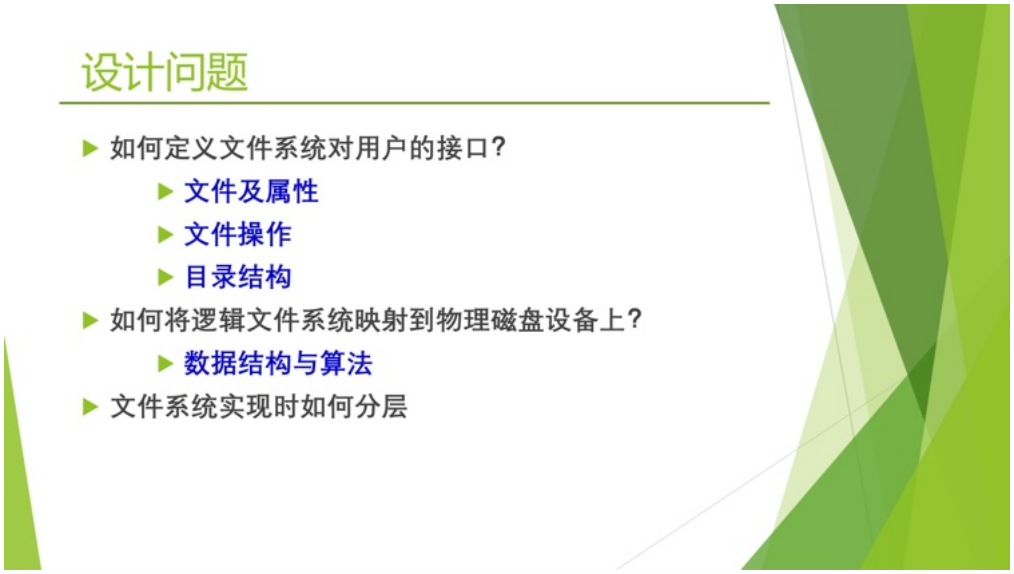
- RAID 0：分条，提高吞吐，无冗余。
- RAID 1：镜像，可靠性高，容量利用率低。

- RAID 4：独立校验盘，校验盘可能瓶颈。
- RAID 5：分布式校验，均衡负载。
- RAID 6：双校验，容忍两盘故障。

> **思考题：** RAID 中数据如何组织存储?

解答： RAID 把逻辑数据切成条带（stripe），分布到多个磁盘上。读写时可以并行访问多个盘，提高吞吐；同时通过镜像或奇偶校验保存冗余信息，在单盘或多盘故障时重建数据。

8.7 文件系统结构



图示：文件系统接口、逻辑文件系统、文件组织模块、基本 I/O 控制和设备驱动分层。

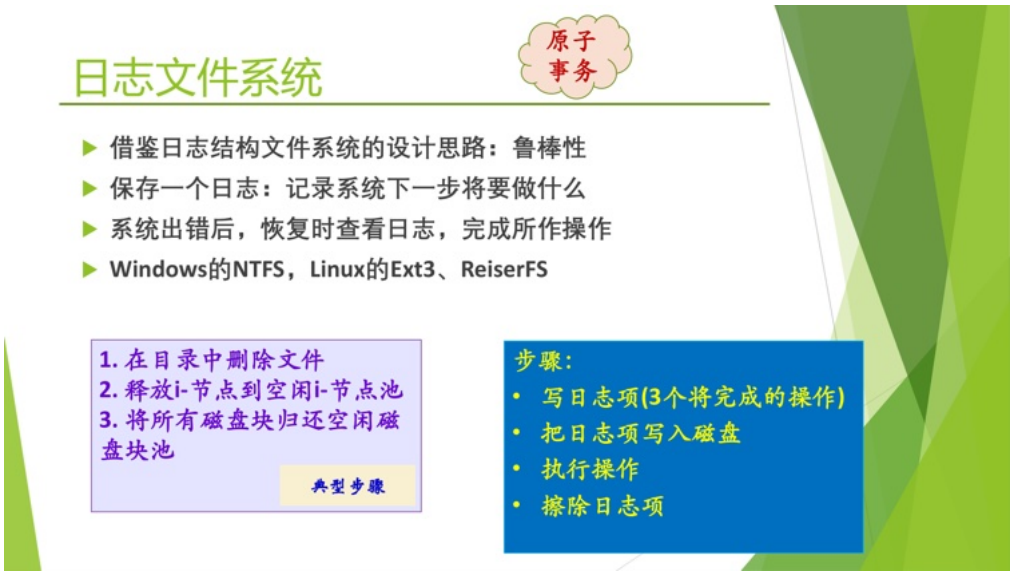
典型层次：

1. 用户接口：`open/read/write/close`；
2. 逻辑文件系统：路径名、目录、权限、元数据；
3. 文件组织模块：逻辑块到物理块映射；
4. 基本 I/O 控制：块缓存、请求队列；
5. 设备驱动与中断：与具体磁盘控制器交互。

8.7.1 VFS

- **虚拟文件系统（Virtual File System, VFS）**为不同文件系统提供统一接口。
- 关键对象包括 `super_block`、`inode`、`dentry`、`file`。
- 每种真实文件系统注册自己的操作表。
- VFS 让 ext4、FAT、procfs、tmpfs、NFS 等都能通过统一系统调用访问。

8.7.2 LFS 与日志文件系统



图示：LFS 把写操作顺序追加到日志中，减少寻找写入位置的开销。

- **LFS（Log-structured File System）**：把所有修改顺序写入日志段，优化写性能。
- **Journaling File System**：重点记录元数据事务，保证崩溃恢复。
- 二者都使用“日志”思想，但目标不同：LFS 以写性能为核心，Journaling 以一致性恢复为核心。

8.8 XV6 日志系统



图示：XV6 没有活跃文件系统调用时才提交日志，保证日志不包含未提交写入。

> **代码阅读题：** XV6 为什么需要 `begin_op` 和 `end_op`?



图示：XV6 commit 将日志块写入真实位置，再清除日志头。

解答：文件系统调用可能修改多个块，必须被日志系统作为事务处理。begin_op 在操作开始前预留日志空间，若日志空间不足或正在提交则等待；end_op 在操作结束时减少活跃操作计数。如果没有活跃操作，最后一个 end_op 会触发 commit。

> **代码阅读题：** XV6 commit 的流程是什么？

解答：

1. 把修改过的缓存块复制到日志区；
2. 写日志头，表示事务提交点；
3. 把日志区内容安装到真实磁盘位置；
4. 清空日志头；
5. 崩溃恢复时若看到非空日志头，就重做安装过程。

> **代码阅读题：** recover_from_log 做什么？

解答：启动时读取日志头，若发现有已提交事务，就根据日志头记录的块号把日志区数据复制回真实位置，然后清空日志。这保证“已提交事务一定生效，未提交事务不会半生效”。

8.9 分布式文件系统

8.9.1 Ceph

- 面向云存储、大数据和高性能计算。
- 强调容错和海量数据管理。
- 通过对象存储、CRUSH 算法和多副本/纠删码实现扩展性和可靠性。



图示：分布式文件系统强调资源、任务、功能和控制的分布，以及对用户隐藏内部细节。

8.9.2 HDFS



图示：HDFS 使用大块存储和副本机制，适合大文件顺序读写。

- 把文件切成大块；
- 多副本分布到不同 DataNode；
- NameNode 管理命名空间和块映射；
- 适合大文件、高吞吐、顺序访问；
- 不适合大量小文件和低延迟随机写。

8.10 本讲小结

- 文件系统可靠性靠坏块管理、备份和恢复。
- 一致性问题来自“多块更新不是天然原子操作”。

- `fsck` 通过重建引用计数修复不一致，日志通过事务避免慢速全盘检查。
- 安全性依赖身份认证和访问控制，ACL 与 Capability 分别以资源和主体为中心。
- 性能优化围绕减少磁盘 I/O：缓存、预读、调度、优化布局、RAID。
- VFS 统一不同文件系统接口，日志和分布式文件系统解决现代系统的可靠性和规模问题。

LECTURE 9

第 9 讲

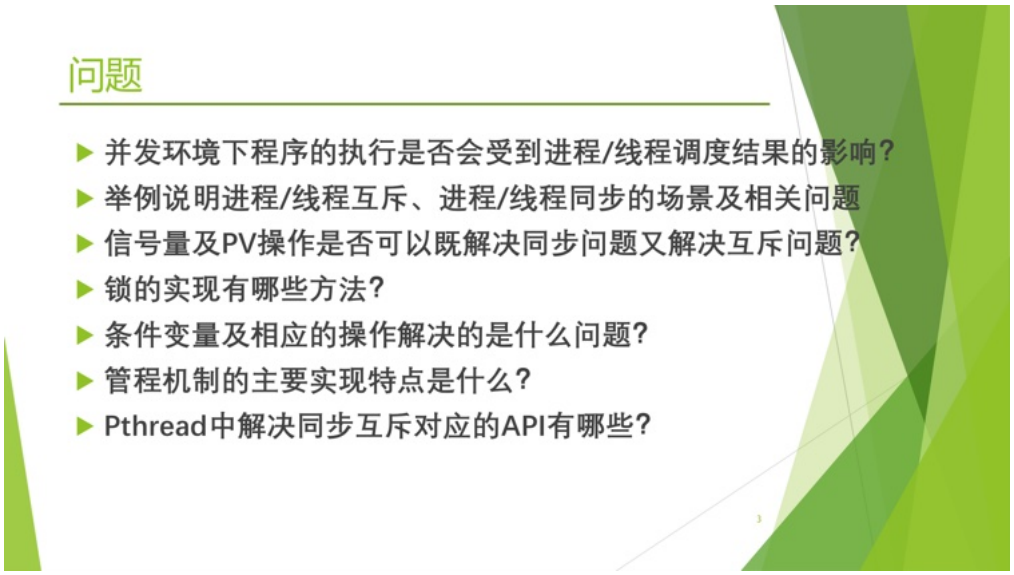
同步机制

Operating Systems Course Notes

9.1 本讲导读

- 本讲从并发执行的不确定性出发，解释为什么需要互斥、同步、信号量、管程和锁。
- 学习主线：
  1. 并发程序为什么会出错；
  2. 竞争条件、临界资源、临界区；
  3. 软件互斥算法与硬件原子指令；
  4. 信号量和 P/V 操作；
  5. 生产者-消费者、读者-写者；
  6. 管程、条件变量、Hoare/Mesa 语义；
  7. Pthreads、Futex、现代硬件原子操作；
  8. XV6 中的 spinlock、sleeplock、sleep/wakeup、pipe。

9.2 开篇问题



图示：并发环境下的调度影响、互斥与同步、信号量、锁、条件变量、管程和 Pthreads API。

> **思考题：**并发环境下程序执行是否会受到进程/线程调度结果的影响?

解答：会。并发程序的指令可以以许多合法交错方式执行。若共享数据访问没有同步，最终结果可能取决于具体交错顺序，这就是竞争条件。例如两个线程同时执行 `cnt++`，每个线程都可能读到旧值，最终只增加一次。

> **思考题：**举例说明互斥和同步的场景。

解答：

- 互斥：两个线程不能同时修改同一个链表；两个进程不能同时打印到同一台独占打印机。
- 同步：消费者必须等缓冲区非空才能消费；父进程必须等子进程退出才能回收；读者写者需要按读/写规则协作。

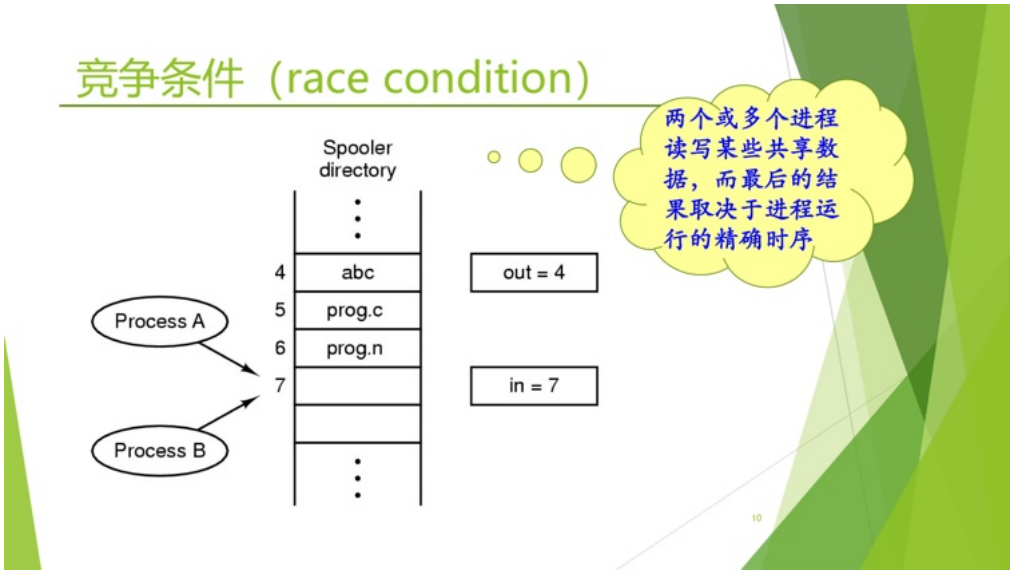
> **思考题：** 信号量及 P/V 操作是否既能解决同步又能解决互斥？

解答： 可以。初值为 1 的二元信号量可作为互斥锁；初值为 0 或资源数量的计数信号量可表达“等待事件发生”或“等待资源可用”。关键是明确信号量在计数什么。

> **思考题：** 条件变量解决什么问题？

解答： 条件变量解决“线程在持有互斥保护的数据结构时，发现条件不满足，需要原子地释放锁并睡眠，醒来后重新检查条件”的问题。它本身不保存资源数量，因此必须和互斥锁、共享状态一起使用。

9.3 并发与竞争条件



图示：cnt++ 被拆成 load、update、store 后，线程交错可能导致错误结果。

9.3.1 与时间有关的错误

- 并发执行是间断性的：进程/线程可能随时被抢占。
- 相对执行速度不可预测：取决于调度、负载、中断、多核。
- 结果可能不确定：同一程序多次运行结果不同。

典型错误：

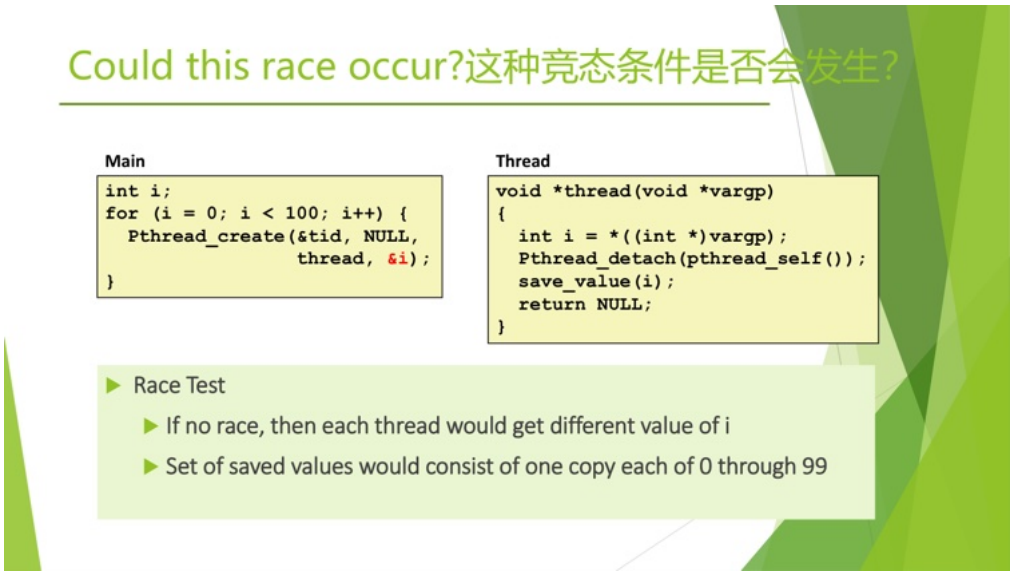
```
for (i = 0; i < niters; i++)
    cnt++;
```

`cnt++` 不是一个原子动作，通常包含：

1. 从内存读 `cnt` 到寄存器；
2. 寄存器加 1；
3. 写回内存。

两个线程交错执行时可能都读取旧值，导致丢失更新。

9.3.2 线程参数传递竞态



图示：循环中把同一个局部变量地址传给多个线程会产生竞态。

> **思考题：** `pthread_create(&tid, NULL, thread, &i)` 中的竞态会发生吗？

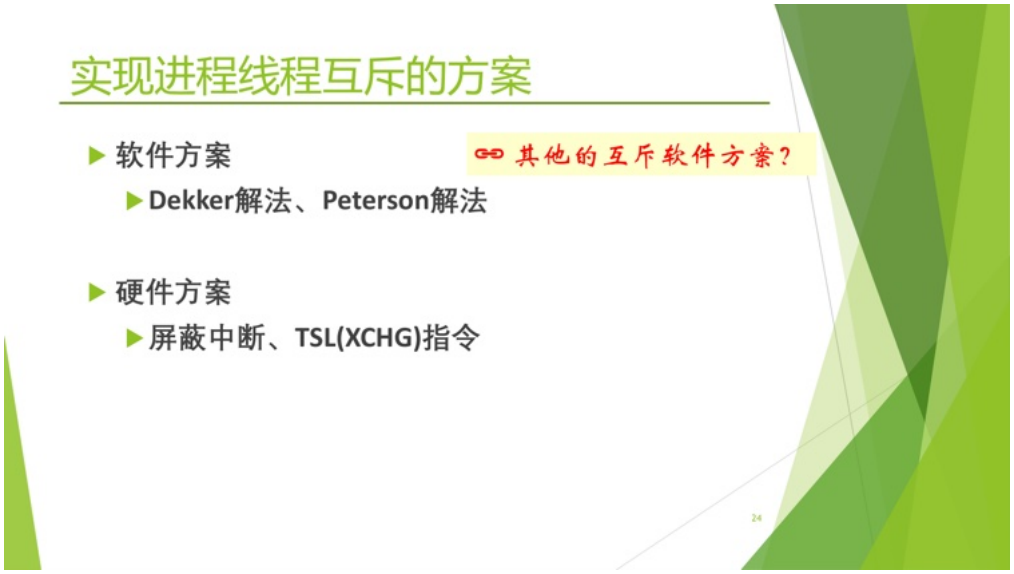
解答： 会。所有新线程拿到的是主线程栈上同一个变量 `i` 的地址。主线程循环继续推进后，子线程解引用时看到的可能已经不是创建时的值，甚至多个线程读到同一个值。正确做法是为每个线程分配独立参数对象，或安全地按值传递。

```c
for (int i = 0; i < 100; i++) {
    int *arg = malloc(sizeof(int));
    *arg = i;
    pthread_create(&tid, NULL, thread, arg);
}
```

9.4 互斥与临界区

9.4.1 基本概念

- **临界资源（Critical Resource）**：一次只允许一个执行单元访问的资源。
- **临界区（Critical Section）**：访问临界资源的代码段。



图示：互斥可由软件算法、屏蔽中断和原子指令实现。

- **互斥（Mutual Exclusion）**：一个执行单元在临界区时，其他执行单元不得进入同一临界区。

临界区原则：

1. 有空让进：临界区空闲时应允许请求者进入；
2. 无空等待：临界区被占用时其他请求者等待；
3. 有限等待：不能让某个请求者无限期等待；
4. 让权等待：等待时最好释放 CPU，而不是一直忙等。

9.4.2 软件互斥算法

- 单标志法：检查和设置不是原子操作，可能两个进程同时进入。
- 强制轮流法：违反有空让进。
- 双标志法：可能双方都表示想进入，导致死锁。
- Dekker 算法：第一个正确的软件互斥算法，但复杂。
- Peterson 算法：用 `interested[]` 和 `turn` 实现两个进程互斥。

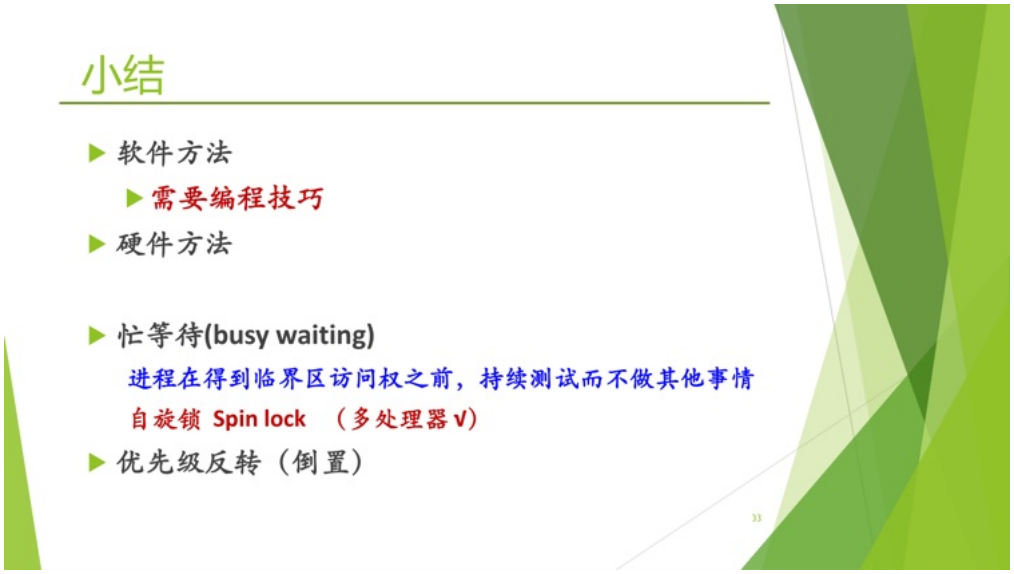
> **思考题：** Peterson 算法满足哪些临界区原则?

解答：对两个进程，在顺序一致内存模型下，Peterson 算法满足互斥、有空让进和有限等待。它不满足让权等待，因为等待进程在 while 循环中忙等。现代弱内存模型下还需要内存屏障，否则编译器或硬件重排可能破坏算法假设。

9.4.3 硬件方案

关闭中断

- 单处理器内核中，关中断可防止当前 CPU 被中断抢占。



图示：Test-and-Set Lock 用原子读改写完成互斥。

- 多处理器上无效，因为其他 CPU 仍可并发访问共享数据。
- 用户态不能随意关闭中断，否则破坏系统安全和响应性。

TSL / XCHG

- **TSL（Test and Set Lock）**原子地读取锁旧值并把锁置为 1。
- **XCHG** 原子交换寄存器与内存。
- 若旧值为 0，则获得锁；若旧值为 1，则继续等待。

> **思考题：** TSL 对多处理器系统有效吗？为什么？

解答：有效。TSL/XCHG 由硬件保证总线或缓存一致性层面的原子读-改-写，多个 CPU 同时执行时只有一个能观察到旧值为 0 并获得锁。代价是竞争激烈时会忙等并产生缓存一致性流量。

9.5 信号量与 P/V 操作

9.5.1 信号量定义

- **信号量（Semaphore）**是一个整数计数器和等待队列的组合。
- `P(s)`：申请资源，`s--`，若资源不足则阻塞。
- `V(s)`：释放资源或发出事件，`s++`，必要时唤醒等待者。

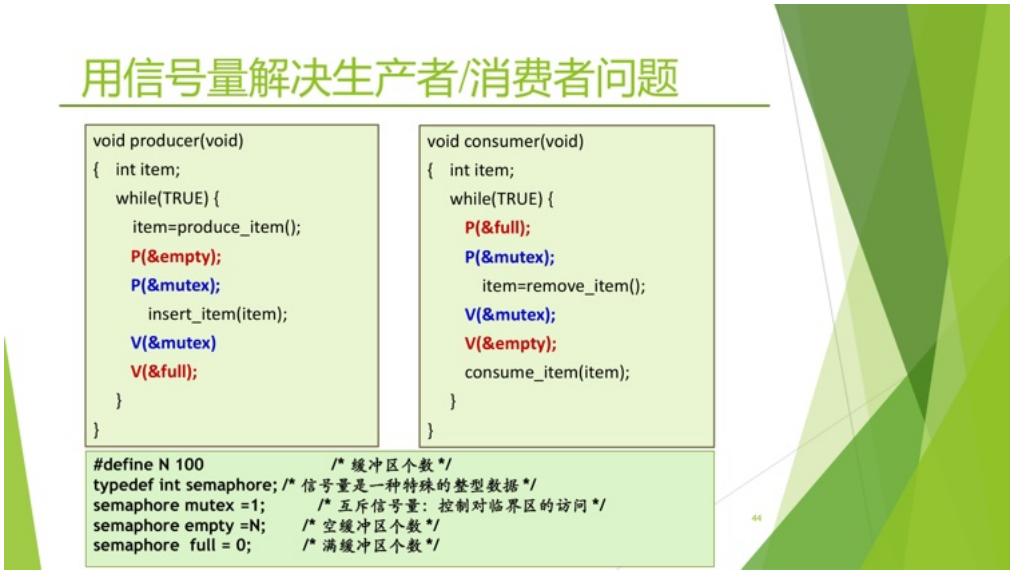
二元信号量用于互斥：

```
P(mutex);
// critical section
V(mutex);
```

计数信号量用于同步：

```
P(full);   // 等待有产品
P(mutex);  // 进入缓冲区临界区
remove();
V(mutex);
V(empty);  // 增加空位
```

9.5.2 生产者-消费者



图示：用 empty、full、mutex 三个信号量解决有界缓冲区问题。

需要解决：

- 缓冲区满时生产者等待；
- 缓冲区空时消费者等待；
- 多个生产者/消费者互斥访问缓冲区。

典型信号量：

- `empty = N`：空槽数量；
- `full = 0`：产品数量；
- `mutex = 1`：互斥访问缓冲区。

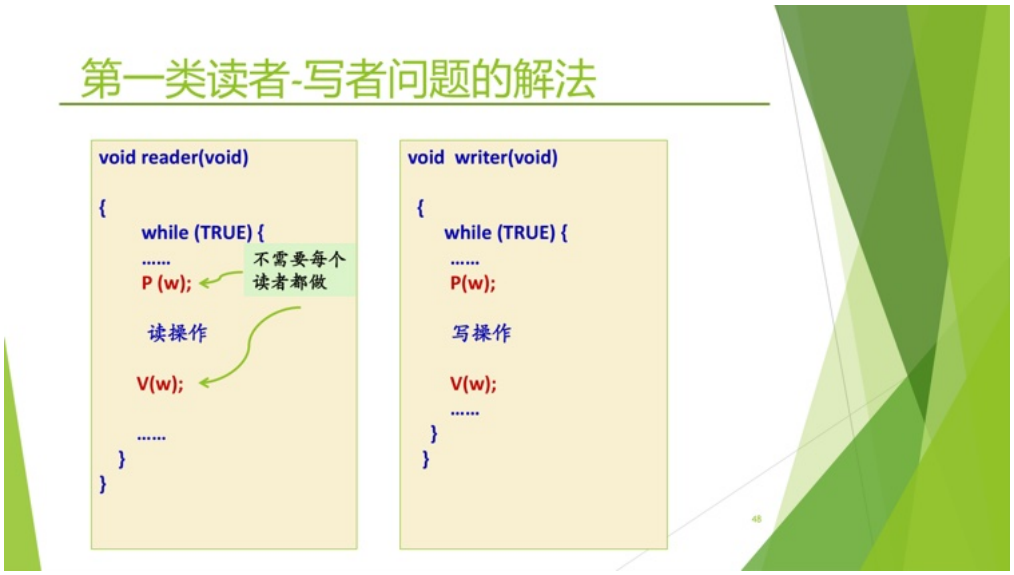
> **思考题：**若颠倒两个 P 操作的顺序会怎样？

解答：若生产者先 `P(mutex)` 再 `P(empty)`，当缓冲区满时，生产者拿着 `mutex` 等 `empty`；消费者需要 `mutex` 才能取走产品并 `V(empty)`，于是死锁。因此必须先等待资源条件，再进入互斥区。

> **思考题：**若颠倒两个 V 操作的顺序会怎样？

解答：一般不破坏正确性，但影响性能和唤醒行为。先 `V(mutex)` 再 `V(full)` 可减少被唤醒消费者马上又因 `mutex` 被占而阻塞的机会；先 `V(full)` 再 `V(mutex)` 也常能工作，但更容易产生不必要调度。

9.5.3 读者-写者



图示：多个读者可并发读，写者必须独占访问共享数据。

第一类读者-写者偏向读者：

- 读者进入时增加 `read_count`；
- 第一个读者锁住写者；
- 最后一个读者释放写者；
- `read_count` 本身必须用 `mutex` 保护。

这体现了一个重要原则：简单资源计数可直接用信号量；复杂状态判断应使用普通变量加锁保护。

9.6 管程与条件变量

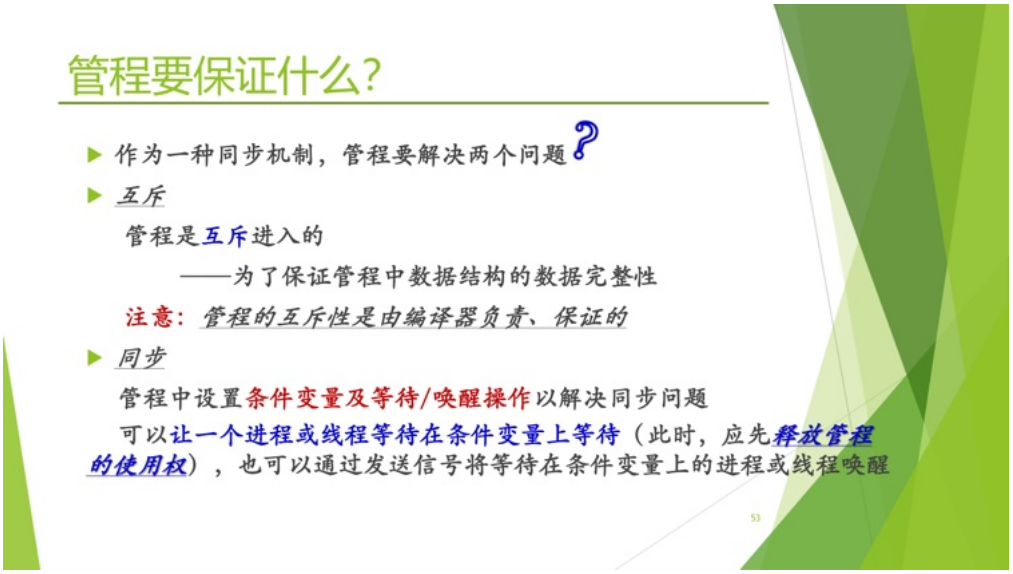
9.6.1 为什么引入管程

信号量的问题：

- P/V 分散在程序各处，难以检查；
- 容易遗漏、颠倒或设错初值；
- 条件复杂时可读性差。

管程的思想：

- 把共享数据封装在一个模块中；
- 外部只能通过管程过程访问；



图示：管程把共享变量和操作封装起来，互斥由管程保证，同步由条件变量实现。

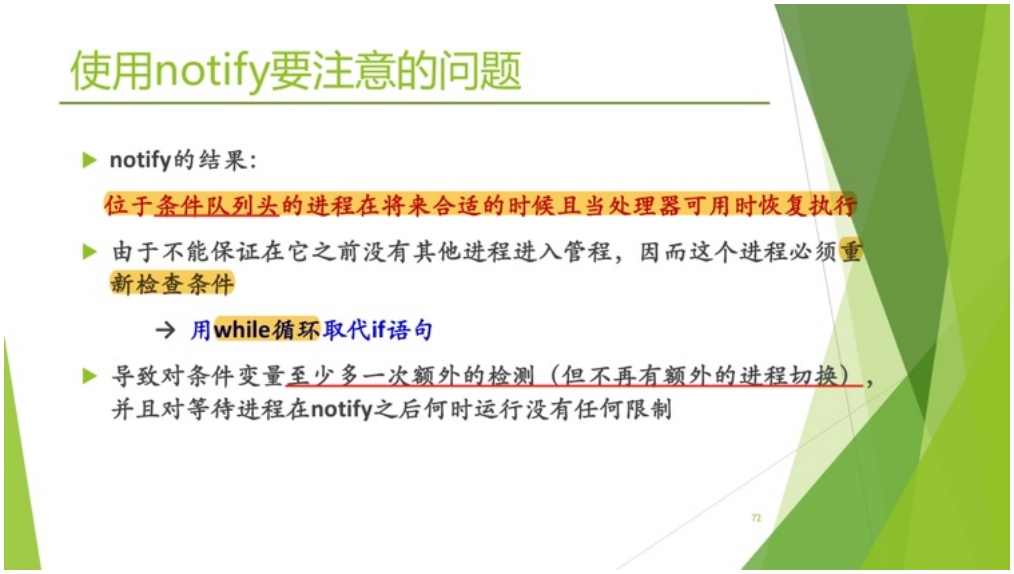
- 同一时刻只有一个线程在管程内；
- 条件变量负责等待和唤醒。

9.6.2 wait / signal



图示：signal 后管程中可能同时存在发信号者和被唤醒者，需要定义语义。

- `wait(c)`：当前线程释放管程锁，在条件队列 `c` 上等待。
- `signal(c)`：唤醒一个等待在 `c` 上的线程。
- `broadcast(c)`：唤醒所有等待者。



图示：Mesa 管程中被通知线程只是进入就绪队列，将来重新竞争管程。

9.6.3 Hoare 与 Mesa 管程

语义	signal 后谁继续运行	优点	注意点
Hoare	被唤醒者立即运行，发信号者进入紧急队列	条件被唤醒时仍成立	实现复杂，多进程切换
Mesa	发信号者继续运行，被唤醒者稍后竞争锁	实现简单，现代系统常用	被唤醒后必须用 `while` 重查条件

> **思考题：** Mesa 管程为什么要用 `while` 而不是 `if` 检查条件?

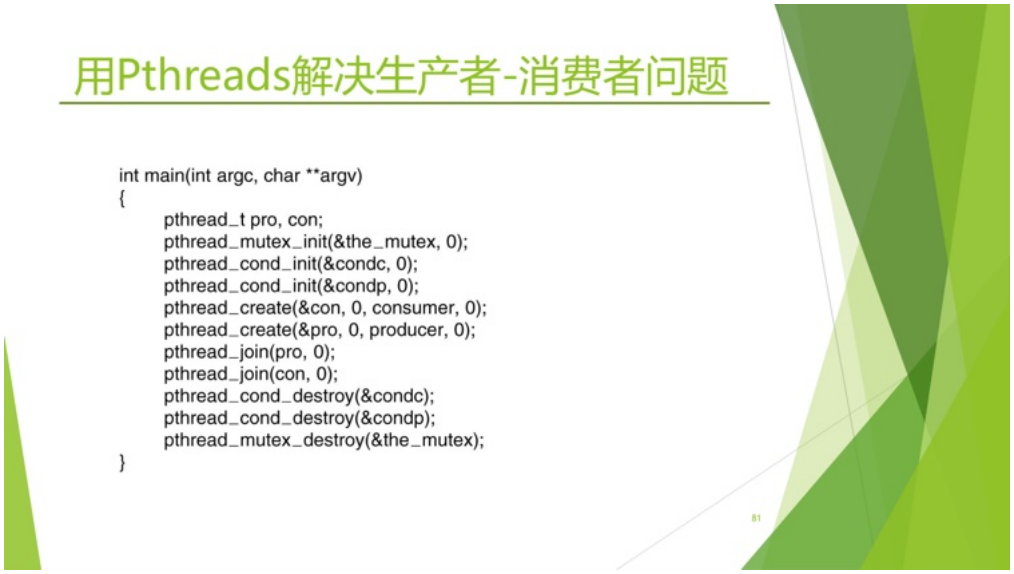
解答： 被 `notify/signal` 唤醒后，线程不会立刻运行，而是重新进入就绪队列。等它真正获得锁时，条件可能已被其他线程改变。因此必须用 `while (!condition) wait(c)`，醒来后重新检查条件。

9.7 Pthreads 同步 API

常用 API:

- `pthread_mutex_init`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`
- `pthread_cond_init`
- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`

条件变量模式:



图示：Pthreads 用 mutex 与 condition variable 实现生产者-消费者。

```c
pthread_mutex_lock(&mutex);
while (!condition)
    pthread_cond_wait(&cond, &mutex);
// condition is true
pthread_mutex_unlock(&mutex);
```

`pthread_cond_wait` 原子地释放互斥锁并睡眠，醒来后重新获得互斥锁再返回。

9.8 锁的实现与现代硬件



图示：简单关闭中断实现锁在多处理器、用户态和长临界区场景都有问题。

> **思考题：**通过开关中断实现锁是否正确?

解答：在单 CPU 内核、短临界区、且调用者可信时可以工作；在多 CPU 上不正确，因为关掉本 CPU 中断不能阻止其他 CPU；在用户态不允许；若临界区很长还会严重影响中断响应。



图示：Futex 使用用户态快速路径和内核态慢路径，兼顾低开销与阻塞等待。

9.8.1 Futex

- 无竞争：用户态通过原子指令快速获得锁。
- 有竞争：进入内核，在 futex 队列上睡眠。
- 解锁时若发现有等待者，调用内核唤醒。
- 解决纯自旋浪费 CPU 和纯内核锁开销大的矛盾。

9.8.2 原子操作与内存顺序

- x86 使用 `LOCK` 前缀实现原子读-改-写。
- RISC-V 使用 LR/SC 和 AMO 指令。
- Acquire/Release 语义控制临界区前后的内存可见性。
- 锁不仅要互斥，还要阻止错误的编译器和硬件重排序。

9.9 XV6 并发同步机制

9.9.1 spinlock 与 sleeplock

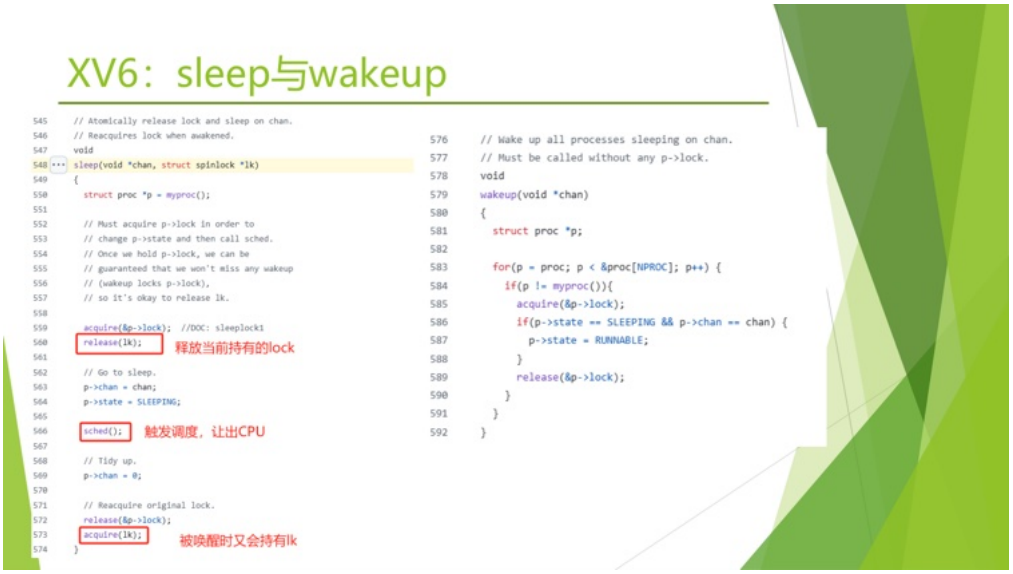
- **spinlock**：
  - 用于短临界区；
  - 持有期间不能睡眠；
  - 常保护进程表、buffer cache 元数据等。
- **sleeplock**：



图示：XV6 提供 spinlock 和 sleeplock，两者适用于不同临界区。

- 获取不到时睡眠；
- 可用于较长 I/O 临界区；
- 常保护具体 buffer 的数据。

9.9.2 sleep / wakeup 与 P/V 类比



图示：XV6 没有直接提供通用信号量，但 `sleep(chan, lock)` 和 `wakeup(chan)` 组合表达等待/唤醒。

> **代码阅读题：** XV6 中 `sleep(chan, lock)` 为什么要传入锁？

解答： 传入锁是为了避免“丢失唤醒”。线程检查条件失败后，必须原子地释放保护条件的锁并进入睡眠。如果先释放锁、再睡眠，中间可能发生 `wakeup`，之后线程再睡就永远醒不过来。XV6 的 `sleep` 在内部接管锁释放和进程状态切换，保证等待条件、释放锁、睡眠之间没有竞态窗口。

9.9.3 XV6 pipe：生产者-消费者



图示：XV6 pipe 使用环形缓冲区实现生产者-消费者，读写双方通过 sleep/wakeup 同步。

> **代码阅读题：** XV6 pipe 如何体现生产者-消费者?

解答：

- `pi->data` 是有限大小环形缓冲区；
- 写者发现缓冲区满时睡在写等待通道上；
- 读者读取数据后唤醒写者；
- 读者发现缓冲区空时睡在读等待通道上；
- 写者写入数据后唤醒读者；
- pipe 锁保护读写下标、打开状态和缓冲区内容。

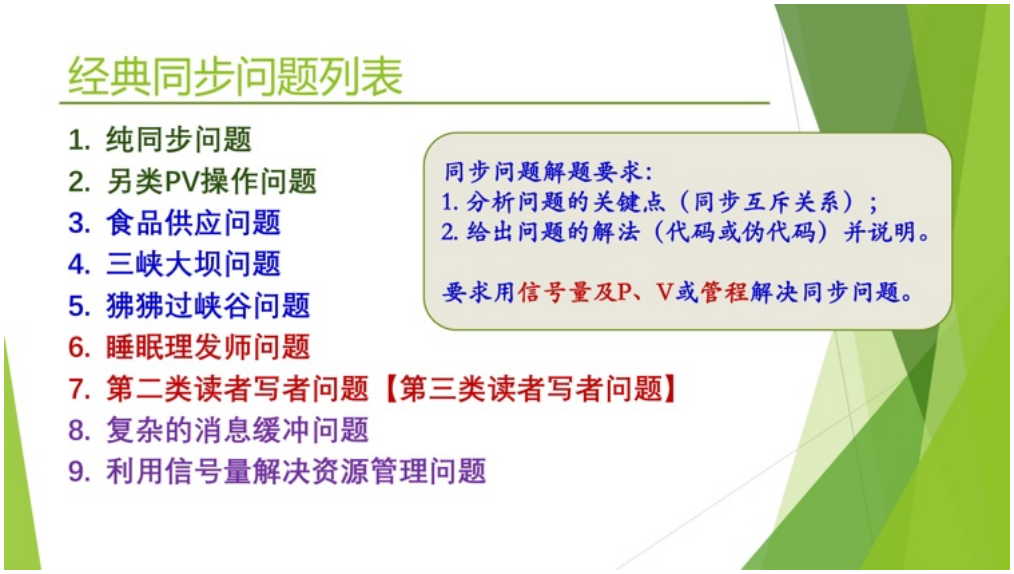
9.10 附录：经典同步问题题型整理

9.10.1 考试解题模板

经典同步题不要一上来写信号量，先把题目翻译成三个层次：

1. **角色**：谁在并发执行，例如生产者、消费者、读者、写者、上行船、东行狒狒、顾客、理发师。
2. **资源**：谁被共享，例如缓冲区、库存计数、绳索、船闸、理发椅、消息缓冲区。
3. **约束**：
   - 互斥：同一时刻谁不能同时做什么；
   - 同步：谁必须等到什么条件成立；
   - 公平：是否要求避免饥饿、是否要求先来先服务、是否要求某类角色优先。

信号量设计时，要反复问一句话：**这个信号量到底在计数什么？**



图示：经典同步问题通常要求先分析同步/互斥关系，再给出 P/V 或管程解法。

- `mutex = 1`：进入临界区的许可；
- `empty = N`：空槽数量；
- `full = 0`：已有产品数量；
- `resource = 1`：资源本身的独占使用权；
- `gate = 1`：某类进程进入系统的大门；
- `turnstile = 1`：公平排队的旋转门。

管程设计时，则把思路换成：

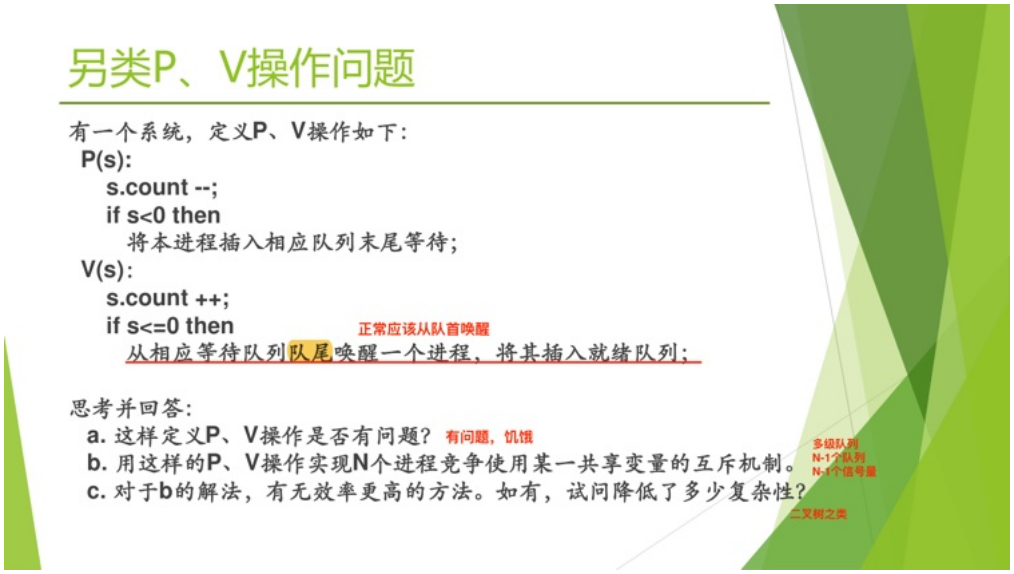
- 共享变量放进管程；
- 每个“等待原因”设置一个条件变量；
- 所有 `wait` 前都用 `while` 检查条件；
- 每次修改状态后，考虑应该 `signal` 还是 `broadcast`。

9.10.2 另类 P/V 操作问题

> **思考题：**这样定义 P/V 是否有问题？如何实现互斥？有没有更高效方法？

解答：

- 互斥仍可实现：用初值为 1 的信号量 `mutex` 包住临界区。
- 问题在公平性：`V` 从队尾唤醒等待者，相当于 LIFO，后来的进程总可能先被唤醒，等待最久的进程可能饥饿。
- 若题目追问“更高效”，重点不是单次队列操作复杂度，而是如何恢复公平：
  - 正常做法：从队头唤醒，FIFO；
  - 若只能使用这种 LIFO 语义，可用多级队列、二叉树队列或额外调度结构维护公平次序，

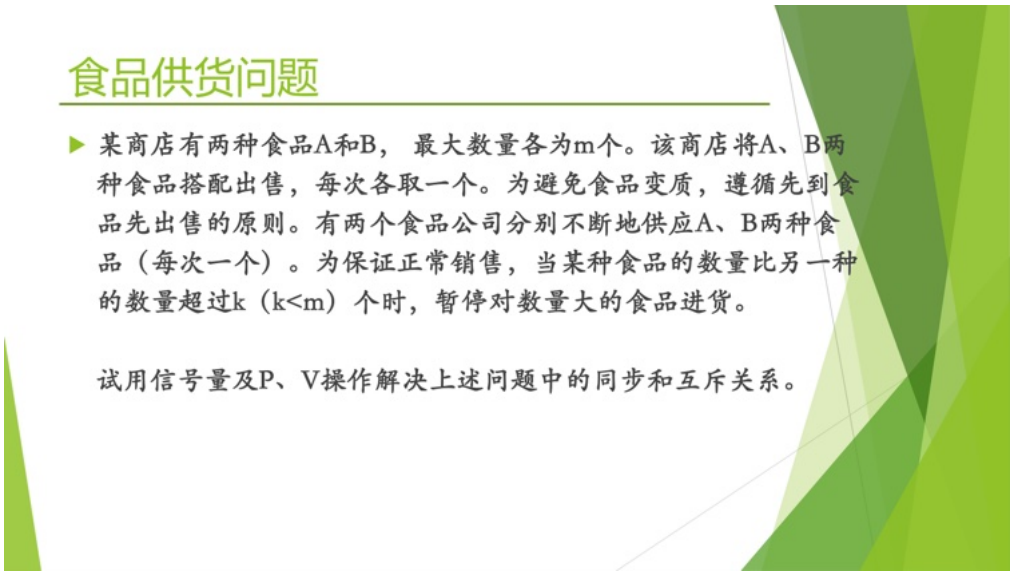


图示：题目把 V 操作改成从等待队列队尾唤醒，会破坏公平性。

但复杂度会上升。

考试要点：**能互斥，不公平；错误性质是饥饿，不是互斥失败**。

9.10.3 食品供货问题



图示：A/B 两种食品容量各为 m，出售必须成对取走，数量差超过 k 时暂停多的一方进货。

关系分析

- 角色：供应 A、供应 B、销售者。
- 共享变量：`a`、`b`，表示当前 A/B 库存。
- 互斥：修改 `a`、`b` 必须互斥。
- 同步：
  - 供应 A 等待 `a < m` 且 `a - b < k`；

- 供应 B 等待 `b < m` 且 `b - a < k`；
- 销售者等待 `a > 0 && b > 0`。

管程解法

```
monitor Store {
    int a = 0, b = 0;
    condition canA, canB, canSell;

    void putA() {
        while (a == m || a - b == k)
            wait(canA);
        a++;
        signal(canSell);
        signal(canB);
    }

    void putB() {
        while (b == m || b - a == k)
            wait(canB);
        b++;
        signal(canSell);
        signal(canA);
    }

    void sell() {
        while (a == 0 || b == 0)
            wait(canSell);
        a--;
        b--;
        signal(canA);
        signal(canB);
    }
}
```

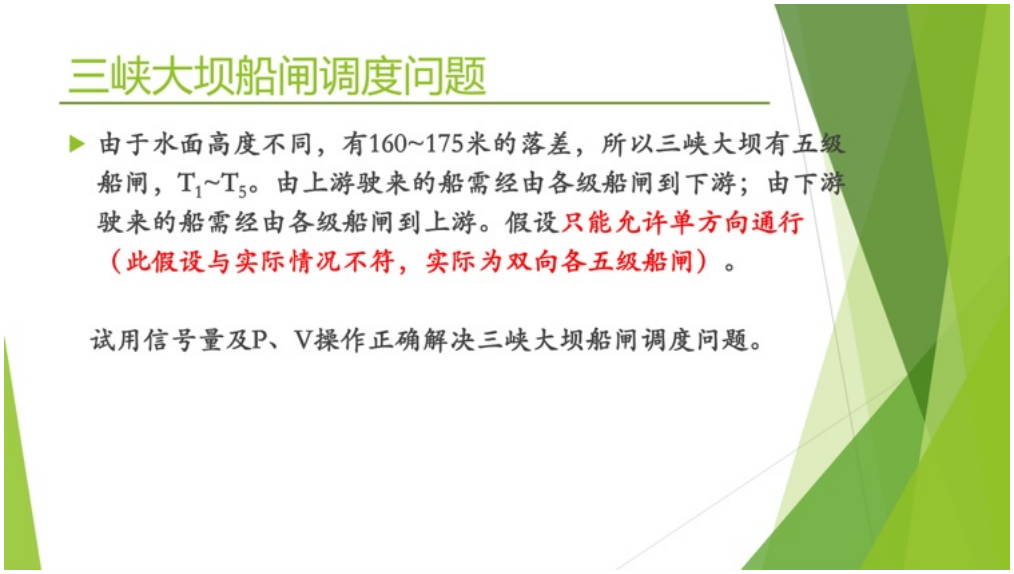
考试易错点

- 只考虑货架满 `a == m`，漏掉数量差 `a - b == k`。
- 销售后只唤醒供应者 A 或 B 中的一方，导致另一方条件已满足但无人唤醒。
- 用 `if` 代替 `while`，Mesa 管程语义下被唤醒后条件可能已经变化。

9.10.4 三峡大坝船闸调度问题

关系分析

- 角色：上行船、下行船。



图示：五级船闸同一时间只允许一个方向通行，每一级船闸还要互斥使用。

- 资源：五级船闸 `T1...T5`。
- 互斥：
  - 每一级船闸一次只能一艘船使用；
  - 整个系统同一时间只能允许一个方向。
- 同步：同方向可流水通过，不同方向必须互斥。

P/V 解法

```
semaphore direction = 1;
semaphore upMutex = 1, downMutex = 1;
semaphore gate[5] = {1, 1, 1, 1, 1};
int upCount = 0, downCount = 0;

void enterUpDirection() {
    P(upMutex);
    if (upCount == 0)
        P(direction);
    upCount++;
    V(upMutex);
}

void leaveUpDirection() {
    P(upMutex);
    upCount--;
    if (upCount == 0)
        V(direction);
    V(upMutex);
}

void enterDownDirection() {
```

```
    P(downMutex);
    if (downCount == 0)
        P(direction);
    downCount++;
    V(downMutex);
}

void leaveDownDirection() {
    P(downMutex);
    downCount--;
    if (downCount == 0)
        V(direction);
    V(downMutex);
}

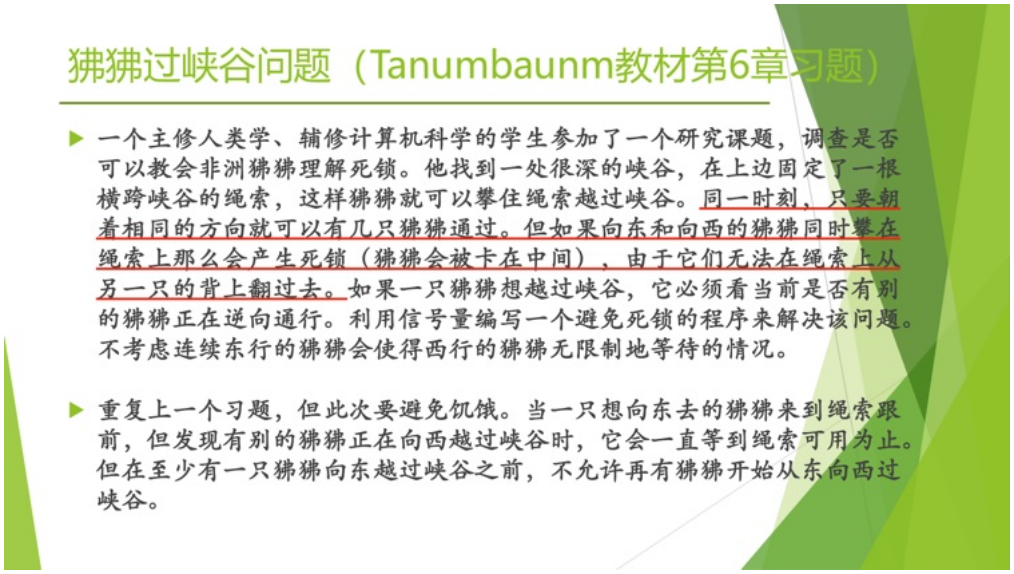
void UpShip() {
    enterUpDirection();
    for (int i = 4; i >= 0; i--) {
        P(gate[i]);
        pass_gate(i);
        V(gate[i]);
    }
    leaveUpDirection();
}

void DownShip() {
    enterDownDirection();
    for (int i = 0; i < 5; i++) {
        P(gate[i]);
        pass_gate(i);
        V(gate[i]);
    }
    leaveDownDirection();
}
```

拓展思考

- 这个题是“两个方向的读者-写者变体”：同方向可并发，不同方向互斥。
- 上述方案可能导致某方向源源不断到来时另一方向饥饿。若题目要求公平，需要加入 `turnstile` 或按批次切换方向。
- 若真实系统允许双向各五级船闸，则模型会变成更复杂的路径资源分配和调度问题，不再是简单方向互斥。

9.10.5 狒狒过峡谷问题



图示：同方向可多只狒狒同时过绳，反方向不能同时在绳上；扩展要求避免饥饿。

不考虑饥饿的 P/V 解法

```
semaphore rope = 1;
semaphore eastMutex = 1, westMutex = 1;
int east = 0, west = 0;

void EastBaboon() {
    P(eastMutex);
    east++;
    if (east == 1)
        P(rope);
    V(eastMutex);

    cross();

    P(eastMutex);
    east--;
    if (east == 0)
        V(rope);
    V(eastMutex);
}

void WestBaboon() {
    P(westMutex);
    west++;
    if (west == 1)
        P(rope);
    V(westMutex);

    cross();

```

```
    P(westMutex);
    west--;
    if (west == 0)
        V(rope);
    V(westMutex);
}
```

这个解法保证不同方向互斥，但不保证公平。例如东行狒狒连续到来时，西行狒狒可能一直等。

避免饥饿的思路

加入 `turnstile`，让想获得方向控制权的狒狒按到达顺序排队：

```
semaphore turnstile = 1;

void EastBaboonFair() {
    P(turnstile);
    P(eastMutex);
    east++;
    if (east == 1)
        P(rope);
    V(eastMutex);
    V(turnstile);

    cross();

    P(eastMutex);
    east--;
    if (east == 0)
        V(rope);
    V(eastMutex);
}
```

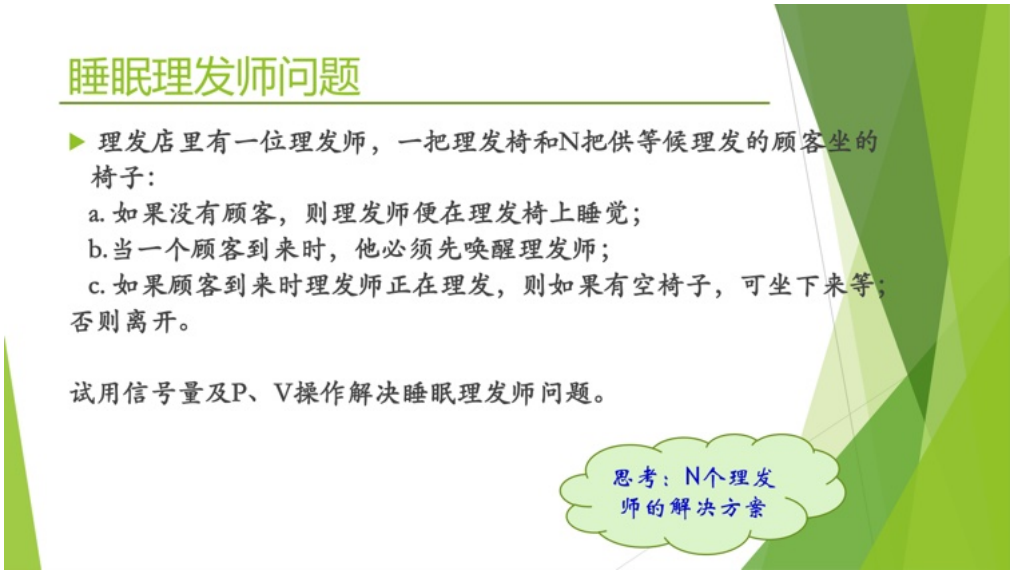
西行对称。关键点是：当西行狒狒已经在 `turnstile` 后等待 `rope` 时，新来的东行狒狒会阻塞在 `turnstile` 外，不能无限插队。

9.10.6 睡眠理发师问题

标准 P/V 解法

```
semaphore mutex = 1;
semaphore customers = 0;
semaphore barbers = 0;
int waiting = 0;
const int N = 5;

void Barber() {
```



图示：无顾客时理发师睡眠；顾客到来时唤醒理发师；等待椅满则离开。

```
    while (true) {
        P(customers);
        P(mutex);
        waiting--;
        V(barbers);
        V(mutex);
        cut_hair();
    }
}

void Customer() {
    P(mutex);
    if (waiting < N) {
        waiting++;
        V(customers);
        V(mutex);
        P(barbers);
        get_haircut();
    } else {
        V(mutex);
        leave();
    }
}
```

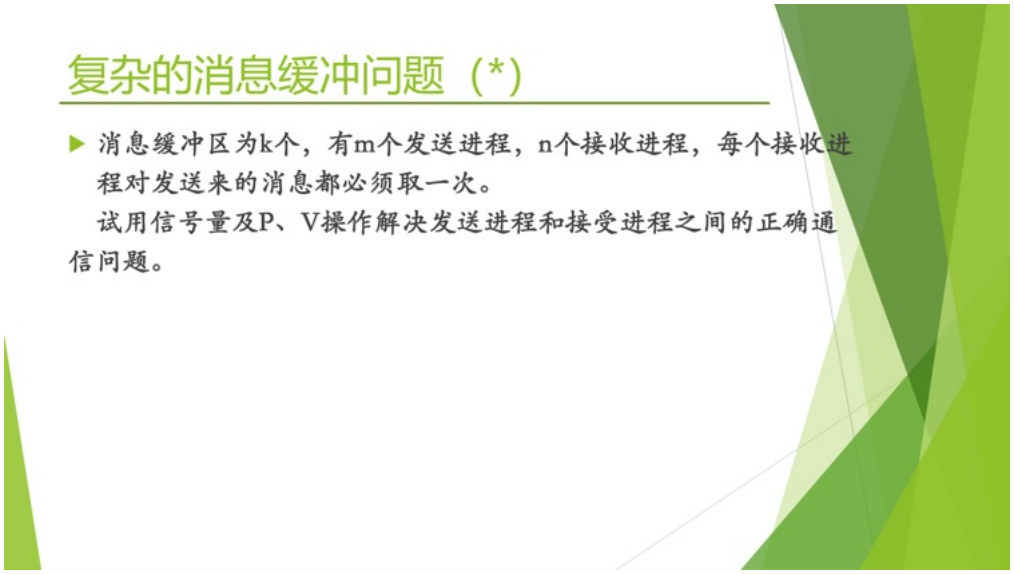
N 个理发师怎么改

- 上述代码本身已经支持多个理发师进程并发执行 `Barber()`。
- `barbers` 表示“可服务顾客的理发师许可”，多个理发师会分别 `V(barbers)`。
- 仍要用 `mutex` 保护 `waiting`。

考试易错点：

- 顾客发现没椅子离开时忘记 `V(mutex)`。
- 理发师从等待区叫走顾客后忘记 `waiting--`。
- 把 `customers` 当普通整数用，而不是信号量，导致“检查为空”和“睡眠”之间出现丢失唤醒。

9.10.7 复杂消息缓冲问题



图示：k 个消息缓冲区，m 个发送进程，n 个接收进程；每条消息必须被每个接收进程各取一次。

关系分析

- 每个发送者生产一条消息。
- 每个接收者都必须读到每条消息一次。
- 一个消息缓冲区不能在所有 `n` 个接收者都读完前回收。
- 需要解决：
  - 空缓冲区管理；
  - 每个接收者的消息队列；
  - 消息引用计数；
  - 最后一个接收者释放缓冲区。

一种可用解法

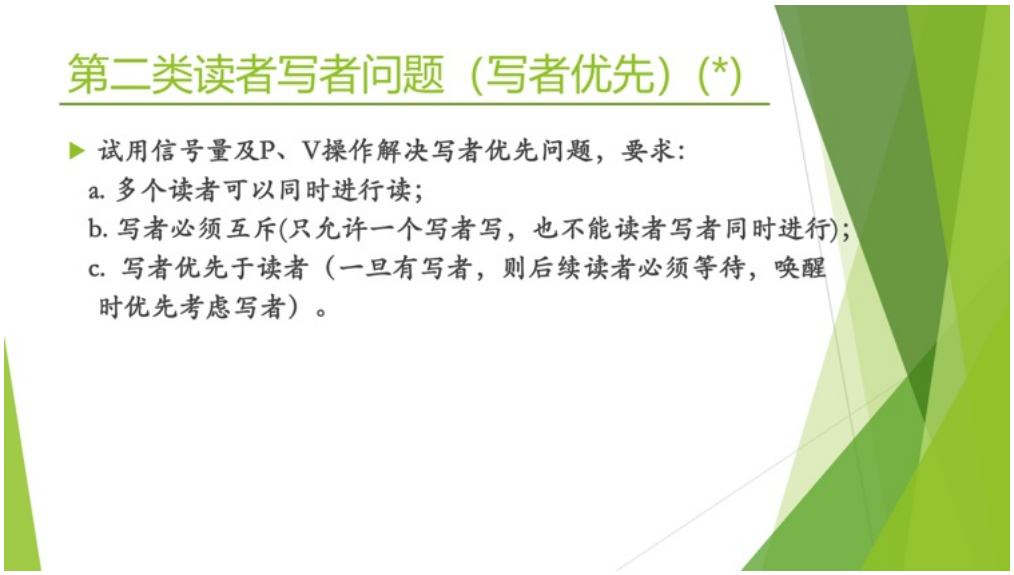
```
semaphore empty = k;
semaphore freeMutex = 1;
semaphore queueMutex[n] = {1, ...};
semaphore msgAvail[n] = {0, ...};
semaphore refMutex = 1;
```

```
6
7  queue freeList;
8  queue recvQ[n];
9
10 struct Msg {
11     char data[MAX];
12     int remain;
13 };
14
15 void Sender() {
16     P(empty);
17     P(freeMutex);
18     Msg *p = freeList.pop();
19     V(freeMutex);
20
21     fill_message(p);
22     p->remain = n;
23
24     for (int i = 0; i < n; i++) {
25         P(queueMutex[i]);
26         recvQ[i].push(p);
27         V(queueMutex[i]);
28         V(msgAvail[i]);
29     }
30 }
31
32 void Receiver(int i) {
33     P(msgAvail[i]);
34     P(queueMutex[i]);
35     Msg *p = recvQ[i].pop();
36     V(queueMutex[i]);
37
38     consume(p);
39
40     P(refMutex);
41     p->remain--;
42     if (p->remain == 0) {
43         P(freeMutex);
44         freeList.push(p);
45         V(freeMutex);
46         V(empty);
47     }
48     V(refMutex);
49 }
```

拓展思考

- 如果每个接收者都必须按发送顺序接收，可以让每个 `recvQ[i]` 保持 FIFO。
- 如果消息很大，不应复制 `n` 份消息体，而应复制 `n` 个指针，配合引用计数。
- `remain` 的修改必须互斥，否则两个接收者可能同时看到 `remain == 1` 或漏掉最后一次释放。

9.10.8 写者优先与公平读者-写者



图示：第二类读者写者要求一旦有写者，后续读者必须等待，唤醒时优先考虑写者。

写者优先 P/V 解法

```
int readCount = 0, writeCount = 0;
semaphore rMutex = 1, wMutex = 1;
semaphore readGate = 1;
semaphore resource = 1;

void Reader() {
    P(readGate);
    P(rMutex);
    readCount++;
    if (readCount == 1)
        P(resource);
    V(rMutex);
    V(readGate);

    read();

    P(rMutex);
    readCount--;
    if (readCount == 0)
        V(resource);
```

```
    V(rMutex);
}

void Writer() {
    P(wMutex);
    writeCount++;
    if (writeCount == 1)
        P(readGate);
    V(wMutex);

    P(resource);
    write();
    V(resource);

    P(wMutex);
    writeCount--;
    if (writeCount == 0)
        V(readGate);
    V(wMutex);
}
```

写者优先的核心是 readGate：第一个等待写者关闭读者大门，让后续读者不能继续进入。

第三类：避免饥饿的读者-写者

考试中若问“解决饥饿”，可以使用公平队列 serviceQueue：

```
semaphore serviceQueue = 1;
semaphore resource = 1;
semaphore rMutex = 1;
int readCount = 0;

void ReaderFair() {
    P(serviceQueue);
    P(rMutex);
    readCount++;
    if (readCount == 1)
        P(resource);
    V(rMutex);
    V(serviceQueue);

    read();

    P(rMutex);
    readCount--;
    if (readCount == 0)
        V(resource);
    V(rMutex);
```

```
}

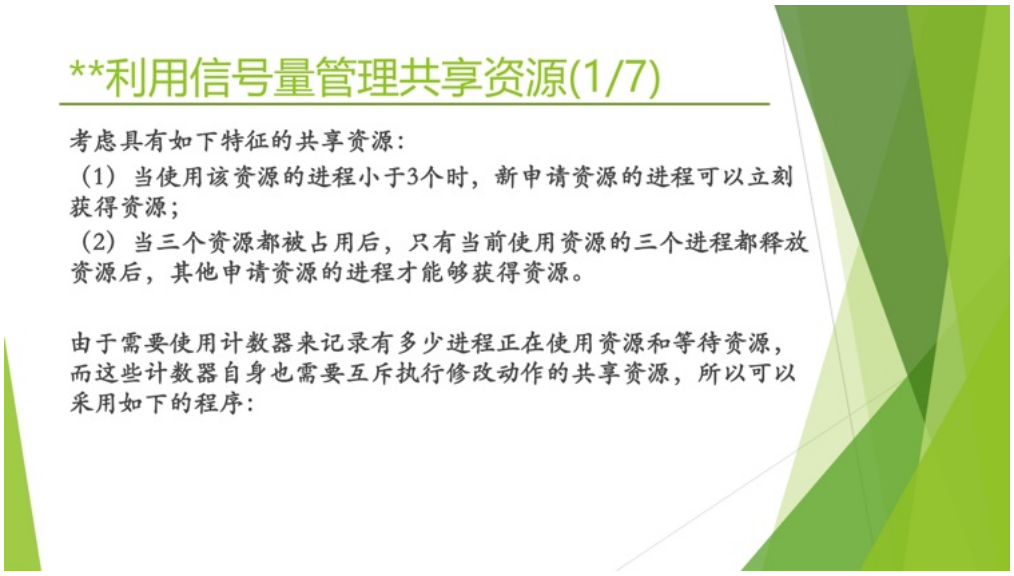
void WriterFair() {
    P(serviceQueue);
    P(resource);
    V(serviceQueue);

    write();

    V(resource);
}
```

serviceQueue 让读者和写者按到达顺序排队，避免读者优先导致写者饥饿，也避免写者优先导致读者长期饥饿。

9.10.9 利用信号量管理共享资源



图示：少于 3 个使用者时可进入；一旦满 3 个，要等当前 3 个全部释放后下一批才能进入。

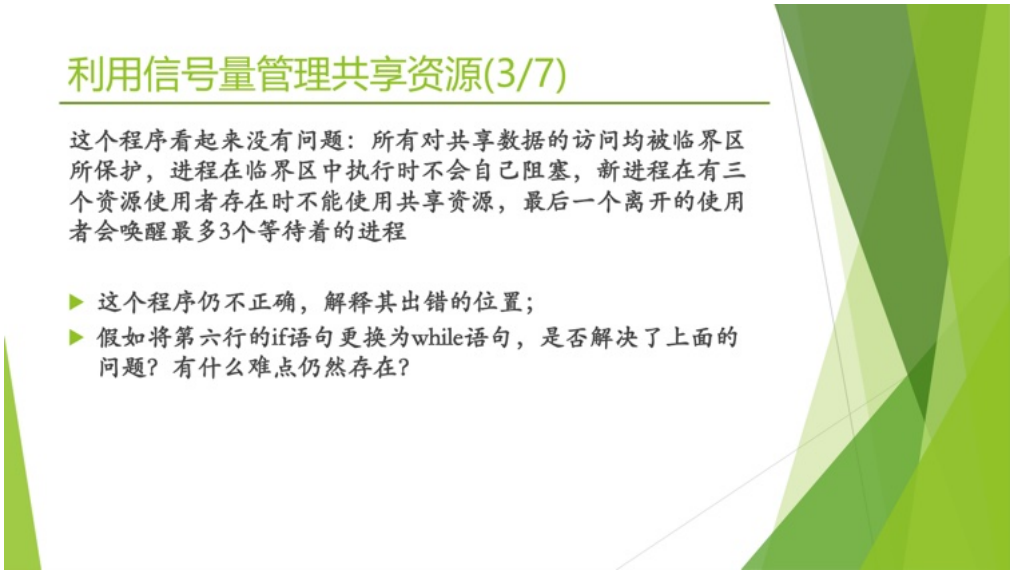
题目规则：

- 同时最多 3 个进程使用资源；
- 一旦达到 3 个，后续进程必须等待；
- 必须等当前 3 个全部释放后，下一批最多 3 个才能进入。

错误程序的问题

错误点在于：等待进程被 V(block) 唤醒后，还要重新竞争 mutex 并自己修改 waiting/active/must_wait。这段时间里新进程可能插入，导致：

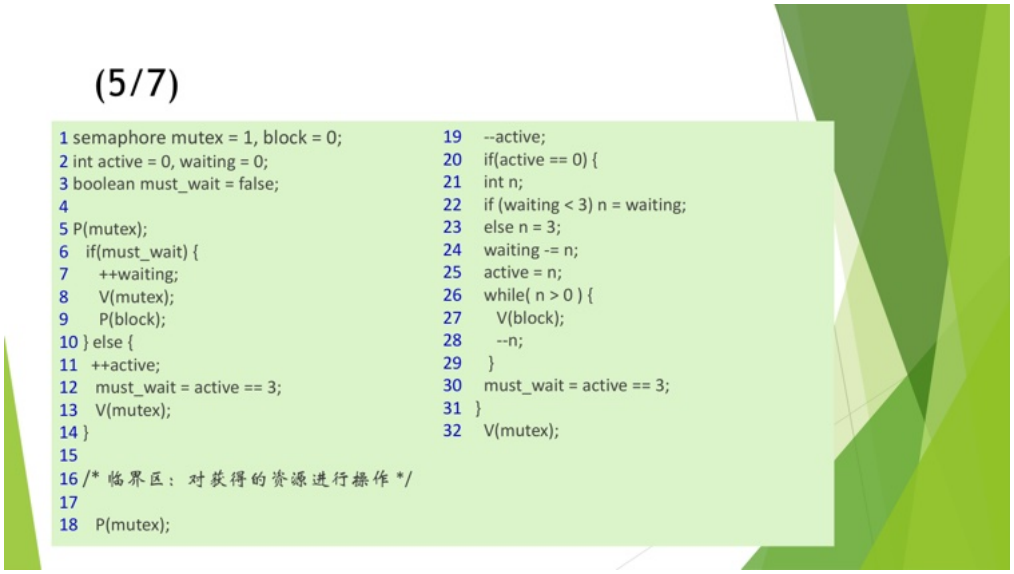
- 被唤醒的进程条件已经变化；
- active 可能超过 3；



图示：错误程序看似所有共享变量都有 mutex 保护，但被唤醒者和新到达者之间仍会竞争。

- 改成 `while` 虽能重新检查条件，但可能出现“已经发出的唤醒被浪费”，进程重新睡下后无人再唤醒。

正确模式一：I'll Do It for You



图示：释放者替被唤醒者完成状态变量修改，被唤醒者醒来后直接使用资源。

```
semaphore mutex = 1, block = 0;
int active = 0, waiting = 0;
bool mustWait = false;

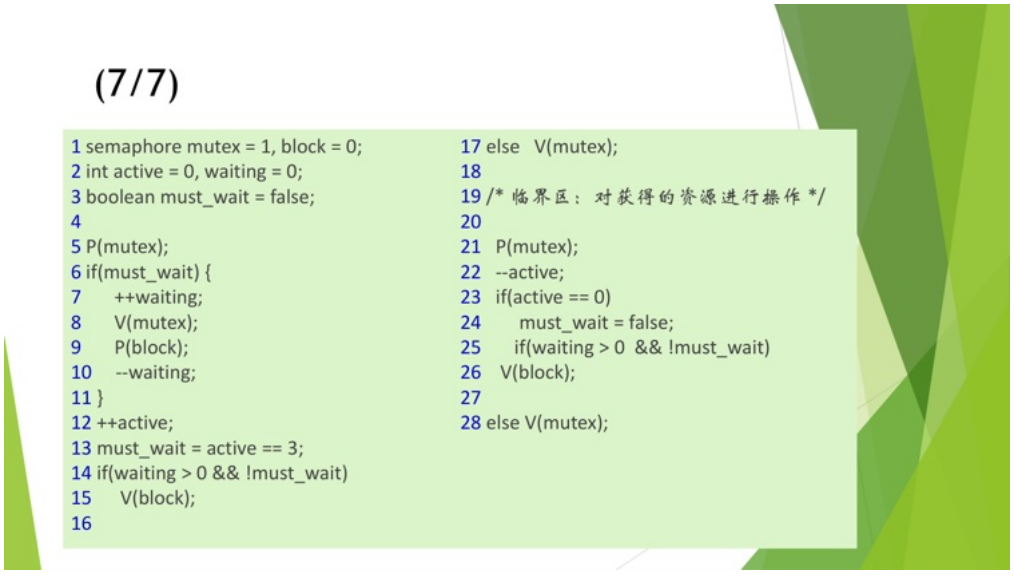
void acquire() {
    P(mutex);
    if (mustWait) {
        waiting++;
        V(mutex);
```

```
        P(block);
    } else {
        active++;
        mustWait = (active == 3);
        V(mutex);
    }
}

void release() {
    P(mutex);
    active--;
    if (active == 0) {
        int n = min(waiting, 3);
        waiting -= n;
        active = n;
        mustWait = (active == 3);
        while (n-- > 0)
            V(block);
    }
    V(mutex);
}
```

核心思想：**释放者替等待者把 active/waiting/mustWait 都改好，再唤醒等待者**。被唤醒者不再参与状态修改，避免醒来后和新进程竞争造成状态错乱。

正确模式二：Pass the Baton



图示：锁的所有权像接力棒一样在等待者之间直接传递，避免被新到达者抢走。

```
semaphore mutex = 1, block = 0;
int active = 0, waiting = 0;
bool mustWait = false;

```

```
void acquire() {
    P(mutex);
    if (mustWait) {
        waiting++;
        V(mutex);
        P(block);
        waiting--;
        // 醒来时默认已经拿到“接力棒”
    }

    active++;
    mustWait = (active == 3);

    if (waiting > 0 && !mustWait)
        V(block);
    else
        V(mutex);
}

void release() {
    P(mutex);
    active--;
    if (active == 0)
        mustWait = false;

    if (waiting > 0 && !mustWait)
        V(block);
    else
        V(mutex);
}
```

核心思想：**V(block) 不只是唤醒等待者，还把进入临界区继续修改状态的权利交给它**。因此被唤醒者不需要重新 P(mutex)，新来的进程也不能插队抢走锁。

9.10.10 高频陷阱总结

- 把“计数变量”误当成“信号量”：普通整数要用 mutex 保护，信号量的 P 操作自带等待语义。
- P 操作顺序写反：例如生产者-消费者中先拿 mutex 再等 empty，可能死锁。
- 忘记最后一个离开者的职责：很多题都靠“第一个进入拿资源，最后一个离开释放资源”。
- 忽略饥饿：读者优先、同方向优先、写者优先都可能让另一类进程长期等待。
- if 代替 while：管程和条件等待中几乎总是应该用 while。
- 唤醒过多但状态没预留：多个进程被唤醒后若还要竞争锁，必须保证状态不会被新进程改变。
- 没有说明正确性：考试写完伪代码后，至少补一句“互斥由哪个信号量保证，同步由哪个条件保证，公平性是否满足”。

9.11 本讲小结

- 并发错误的根源是共享数据访问与执行交错不确定。
- 互斥保护临界区，同步表达执行顺序和条件等待。
- 软件算法展示互斥本质，真实系统主要依赖硬件原子指令。
- 信号量可解决互斥和同步，但复杂状态下容易出错。
- 管程把共享状态、互斥和条件等待封装起来，是现代并发库的核心模型。
- XV6 的锁、sleep/wakeup 和 pipe 是理解操作系统同步机制的好例子。

LECTURE 10

第 10 讲

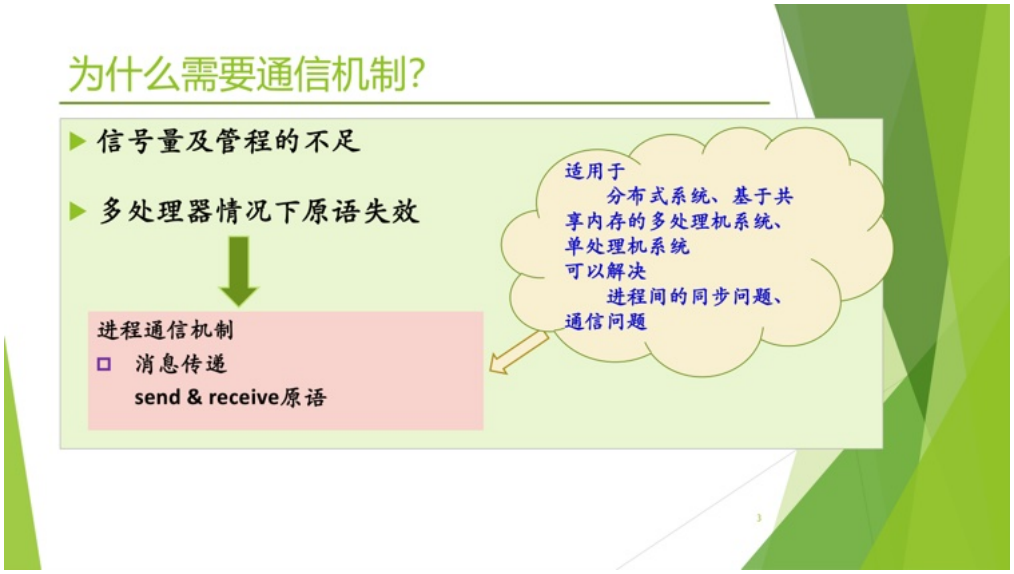
进程间通信

Operating Systems Course Notes

10.1 本讲导读

- 同步机制解决“按什么顺序执行”和“谁能进入临界区”，进程间通信进一步解决“如何交换信息”。
- 本讲主线：
  1. IPC 为什么必要；
  2. 消息传递：`send/receive`；
  3. 共享内存；
  4. 管道、命名管道、套接字、RPC；
  5. Linux IPC 与内核同步机制；
  6. 原子操作、自旋锁、读写锁、RCU、完成变量、屏障、Futex；
  7. 思考题：`receive` 原语、共享内存两个问题、单 CPU 中断并发、屏障的 PV 实现。

10.2 为什么需要 IPC



图示：信号量和管程不足以表达所有通信需求，消息传递可用于单机、多处理器和分布式系统。

- 信号量和管程偏向同步，不擅长传递复杂数据。
- 多处理器和分布式环境下，不同执行单元可能没有天然共享地址空间。
- 进程需要交换数据、通知事件、请求服务、共享结果。
- IPC 机制既能解决通信，也常顺带解决同步。

常见 IPC：

- 管道；
- 消息队列；
- 共享内存；

- 信号量；
- 信号；
- 套接字；
- 文件映射；
- RPC；
- Linux 内核中的自旋锁、读写锁、RCU、Futex 等同步/通信辅助机制。

10.3 消息传递



图示：发送进程通过内核消息缓冲区把消息挂到接收进程消息队列。

10.3.1 基本接口

```
send(destination, message);
receive(source, message);
```

消息通常包含：

- 消息头：类型、接收进程 ID、发送进程 ID、长度、控制信息；
- 消息体：真正的数据内容。

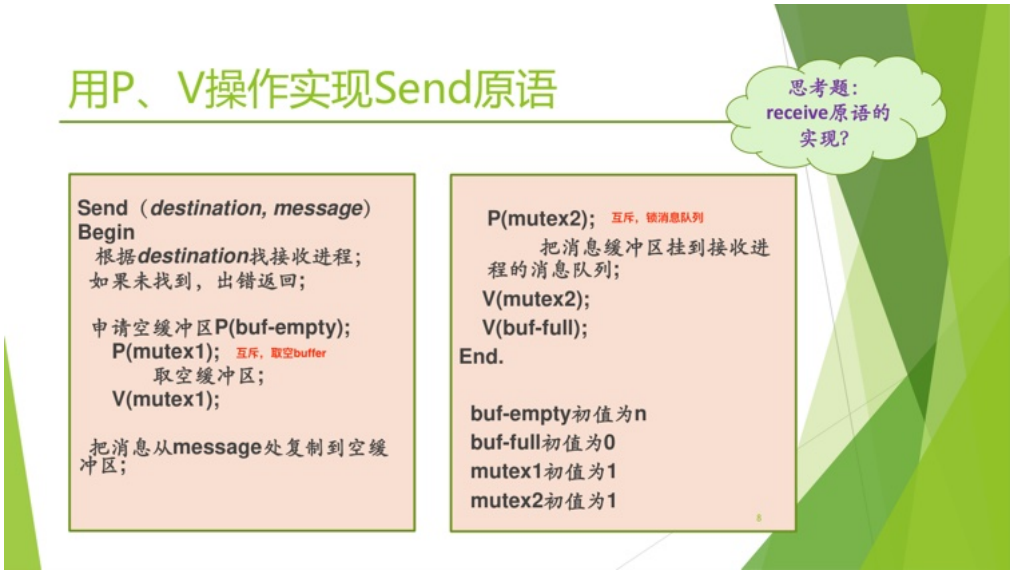
发送流程：

1. 发送者陷入内核；
2. 内核申请消息缓冲区；
3. 从用户地址空间复制消息；
4. 将消息挂入接收者队列；
5. 唤醒可能等待的接收者。

接收流程：

1. 接收者陷入内核；
2. 查找指定来源或任意来源的消息；
3. 若没有消息，根据语义阻塞或返回；
4. 从消息队列取出消息；
5. 复制到用户缓冲区；
6. 释放消息缓冲区并唤醒可能等待空缓冲区的发送者。

10.3.2 用 P/V 实现 send



图示：send 需要空缓冲区、空闲缓冲链表互斥和接收者消息队列互斥。

典型信号量：

- `buf-empty = n`：空消息缓冲区个数；
- `buf-full = 0`：接收队列中消息个数；
- `mutex1 = 1`：保护空缓冲区链表；
- `mutex2 = 1`：保护接收者消息队列。

> **思考题：** `receive` 原语如何实现？

解答：

```
Receive(source, message) {
    P(buf-full);                    // 等待至少有一条消息
    P(mutex2);                      // 互斥访问接收进程消息队列
    从接收进程消息队列中取出一条匹配 source 的消息;
    V(mutex2);

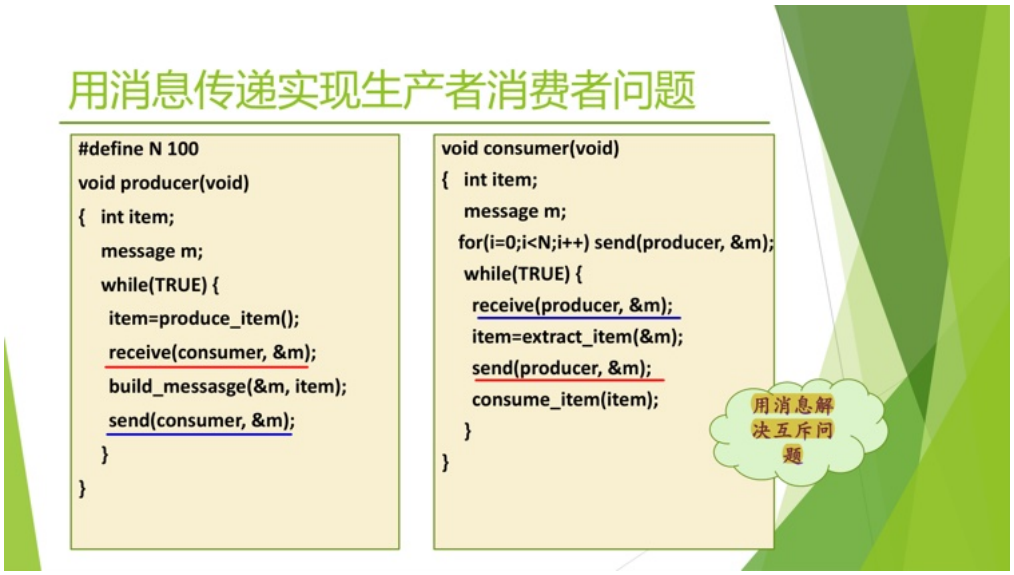
```

```
    将消息内容复制到用户给出的 message 缓冲区;

    P(mutex1);                    // 互斥访问空缓冲区链表
    把消息缓冲区归还空闲链表;
    V(mutex1);
    V(buf-empty);                 // 空缓冲区数量增加
}
```

如果支持按 source 过滤，则 buf-full 只是表示“队列非空”，还需在队列内查找匹配消息；若没有匹配消息，可能要重新睡眠在更细粒度条件队列上。

10.3.3 用消息传递实现生产者-消费者



图示：消费者先发送 N 个空消息作为令牌，生产者收到空消息后填充数据再发回。

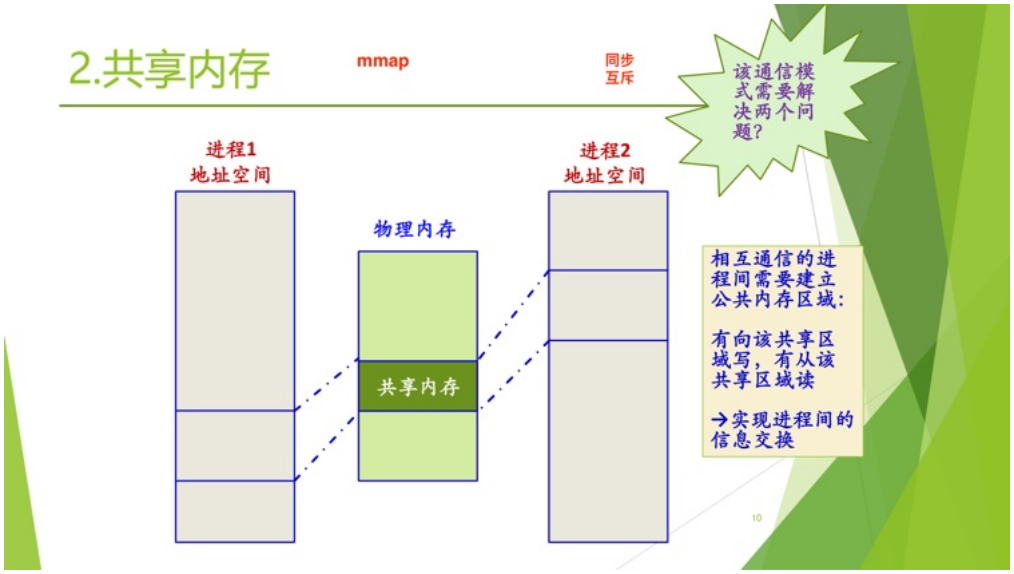
- 消费者先发送 N 个空消息给生产者，表示缓冲区有 N 个空槽。
- 生产者每生产一个 item，先 receive(consumer, &m) 获得空槽令牌，再把 item 放入消息并 send(consumer, &m)。
- 消费者 receive(producer, &m) 获得产品，消费后再把空消息发回。

这个方案把“空槽”和“产品”都表达为消息，避免显式共享缓冲区。

10.4 共享内存

共享内存特点：

- 速度快：通信时不需要每次复制数据到内核消息缓冲区；
- 适合大数据量、高频通信；
- 需要额外同步机制保护共享区；
- 需要建立映射和权限控制。



图示：两个进程把同一物理内存区域映射到各自地址空间，通过读写共享区通信。

> **思考题：**共享内存通信模式需要解决哪两个问题?

解答：

1. **如何建立共享区域**：内核需要把同一物理页框映射到多个进程虚拟地址空间，设置权限，并管理生命周期。
2. **如何同步访问共享区域**：多个进程同时读写会产生竞态，需要用信号量、互斥锁、条件变量、futex 或其他同步原语协调。

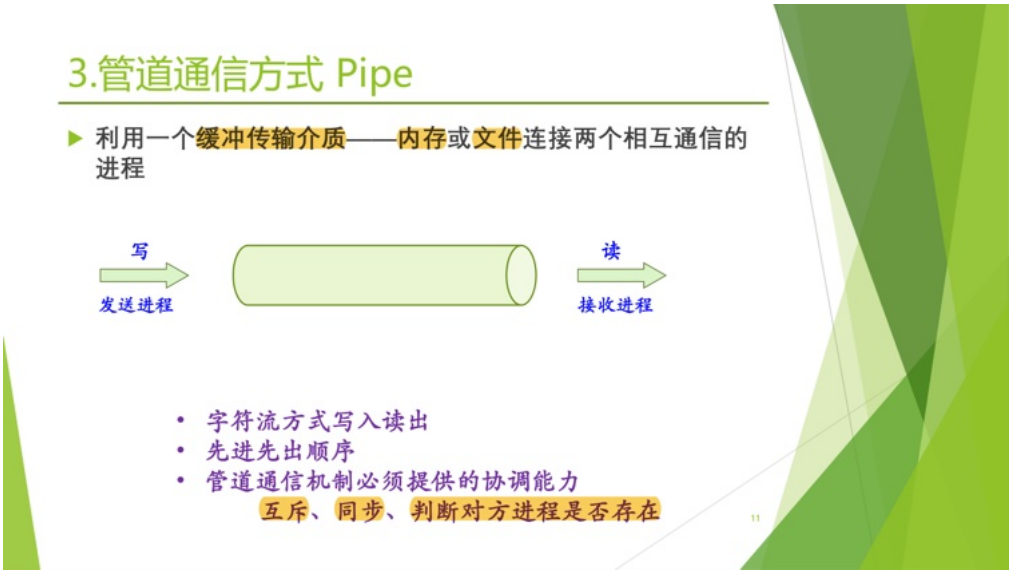
10.5 管道、套接字与 RPC

10.5.1 管道

- 管道提供字节流；
- FIFO 顺序；
- 通常半双工；
- 匿名管道适合父子进程；
- 命名管道可用于无亲缘关系进程；
- 管道必须提供互斥、同步和对端存在性检测。

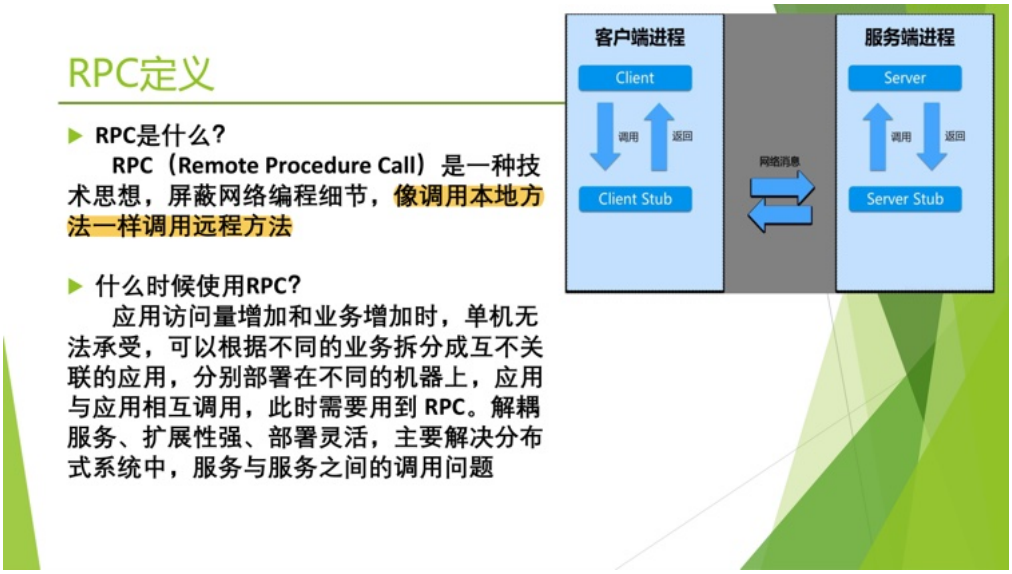
10.5.2 套接字

- 套接字抽象通信端点。
- 可用于本机进程间通信，也可用于网络通信。
- 服务器：`socket`、`bind`、`listen`、`accept`、`recv/send`、`close`。
- 客户端：`socket`、`connect`、`send/recv`、`close`。



图示：管道用内存或文件作为缓冲介质，连接写进程和读进程。

10.5.3 RPC



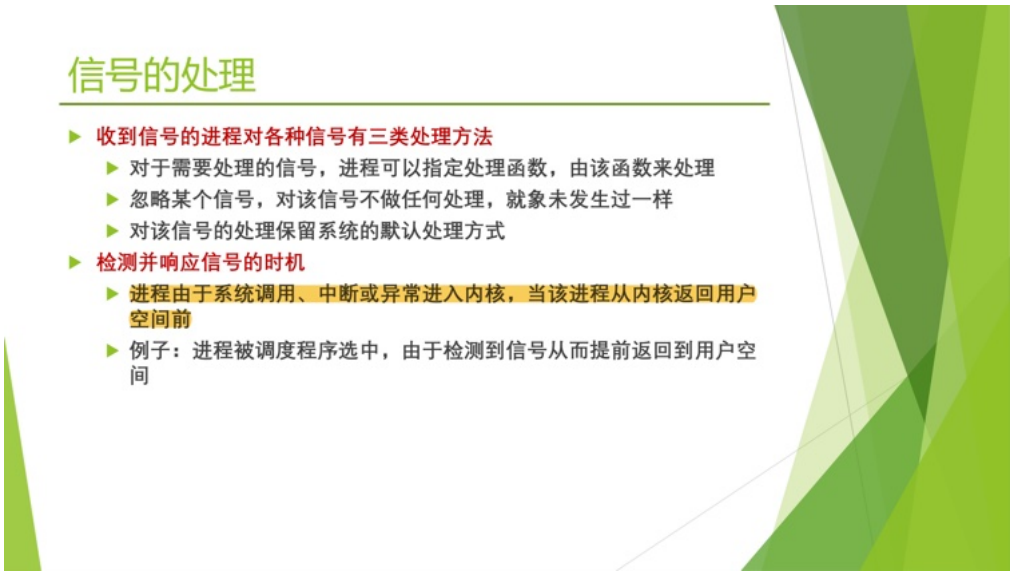
图示：RPC 屏蔽网络细节，让程序像调用本地过程一样调用远程服务。

RPC 需要解决：

- 参数序列化和反序列化；
- 服务发现；
- 网络传输；
- 超时与重试；
- 错误语义；
- 版本兼容；
- 幂等性和事务边界。

RPC 的意义不是让远程调用真的等同本地调用，而是把网络通信封装成更易用的过程调用模型。

10.6 信号



图示：信号是异步通知机制，进程可以注册处理函数。

- 信号用于异步事件通知，如 `SIGINT`、`SIGTERM`、`SIGCHLD`、`SIGALRM`。
- 信号处理函数在进程上下文中异步执行。
- 处理函数应只调用 async-signal-safe 函数。
- 信号可被屏蔽、忽略、捕获或采用默认动作。

Linux 信号处理大致过程：

1. 内核记录待处理信号；
2. 从内核态返回用户态前检查信号；
3. 构造用户栈上的信号处理现场；
4. 跳转到用户注册的 handler；
5. handler 返回后通过 trampoline 恢复原执行流。

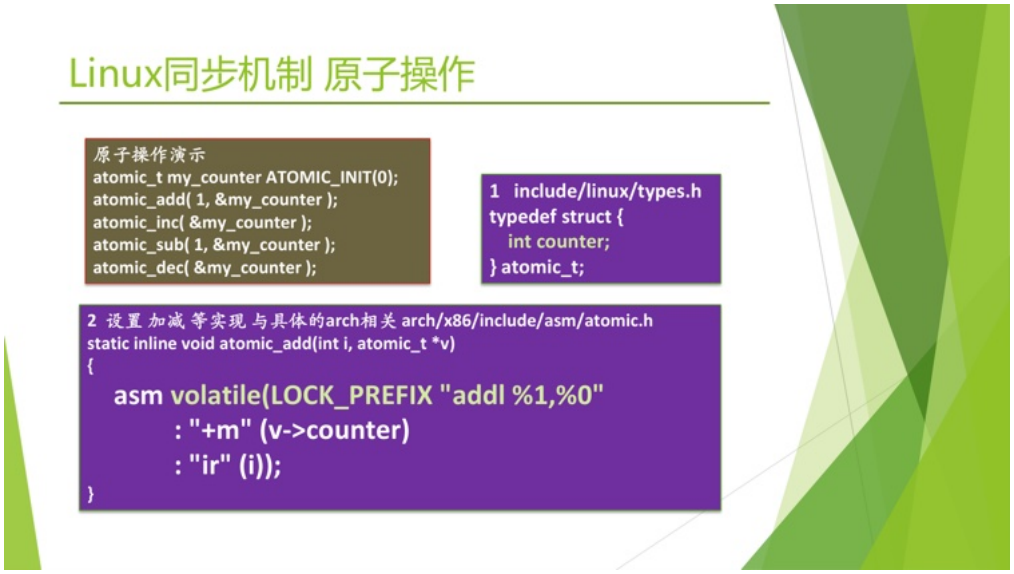
10.7 Linux 内核同步机制

10.7.1 原子操作

- 原子操作不可分割。
- 常用于引用计数、状态位、无锁数据结构的局部步骤。
- 不能替代锁保护复杂临界区。

10.7.2 自旋锁

自旋锁特点：



图示：Linux 原子变量常用于引用计数等简单共享状态。



图示：传统自旋锁忙等获取锁，竞争时消耗 CPU。

- 获取不到时忙等；
- 临界区必须短；
- 持锁期间不能睡眠；
- 适合中断上下文和多核短临界区。

> **思考题：**单 CPU 上如何解决中断处理过程中的并发？

解答：单 CPU 上普通线程与中断处理不会真正并行，但中断可在临界区中打断当前执行流。因此内核可通过关中断或保存/恢复中断状态，防止中断处理程序访问同一共享数据。若涉及进程上下文之间的并发，还需要锁或禁止抢占。

> **思考题：**传统自旋锁实现有什么问题？

解答：简单 test-and-set 自旋锁在竞争激烈时所有 CPU 不断写同一缓存行，导致缓存一致性流量很大；也不保证公平，可能饥饿。票据锁（ticket lock）用“取号 + 叫号”改善公平性，但在大规模多核上仍有共享计数器争用。

10.7.3 自旋锁与信号量



图示：自旋锁忙等，信号量睡眠；选择取决于临界区长度和是否可能睡眠。

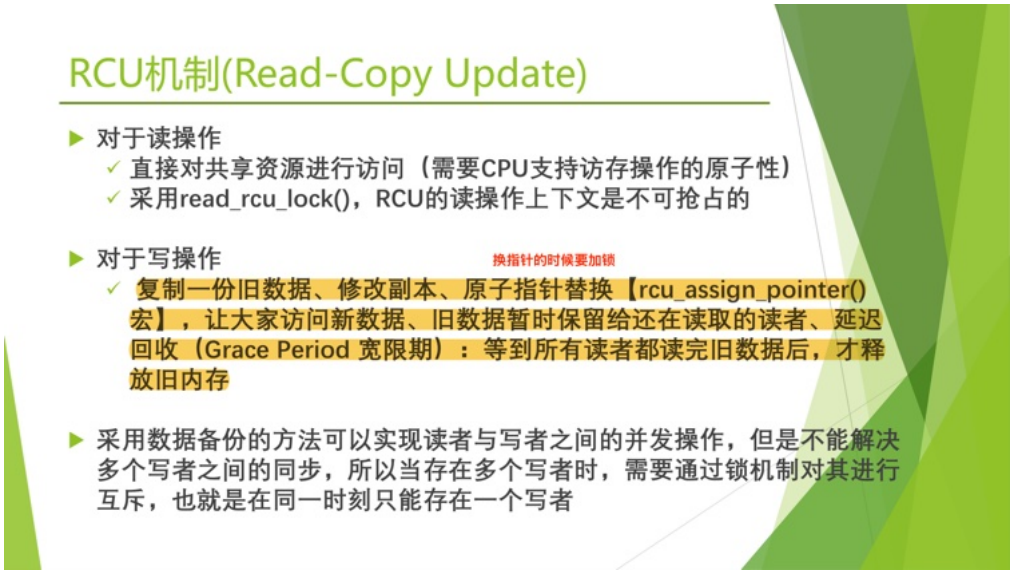
机制	等待方式	适合场景	不适合场景
自旋锁	忙等	短临界区、中断上下文、多核内核同步	可能睡眠、临界区长
信号量/睡眠锁	阻塞睡眠	临界区长、等待 I/O、资源数量管理	中断上下文、极短临界区

10.7.4 读写锁

- 多个读者可同时持有读锁；
- 写者必须独占；
- 读多写少的数据结构收益明显；
- 需考虑写者饥饿或读者饥饿。

10.7.5 RCU

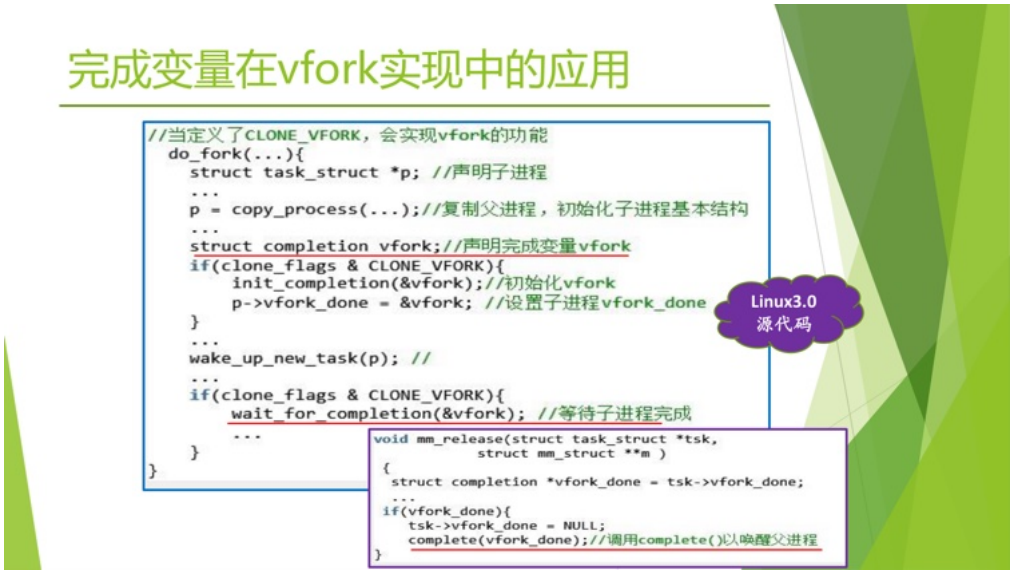
- **Read-Copy-Update**：
  1. 读者无锁读取旧版本；
  2. 写者复制并修改新版本；
  3. 原子发布新指针；
  4. 等待所有旧读者离开宽限期；
  5. 回收旧版本。



图示：RCU 通过读端无锁、写端复制和宽限期回收实现高性能读多写少并发。

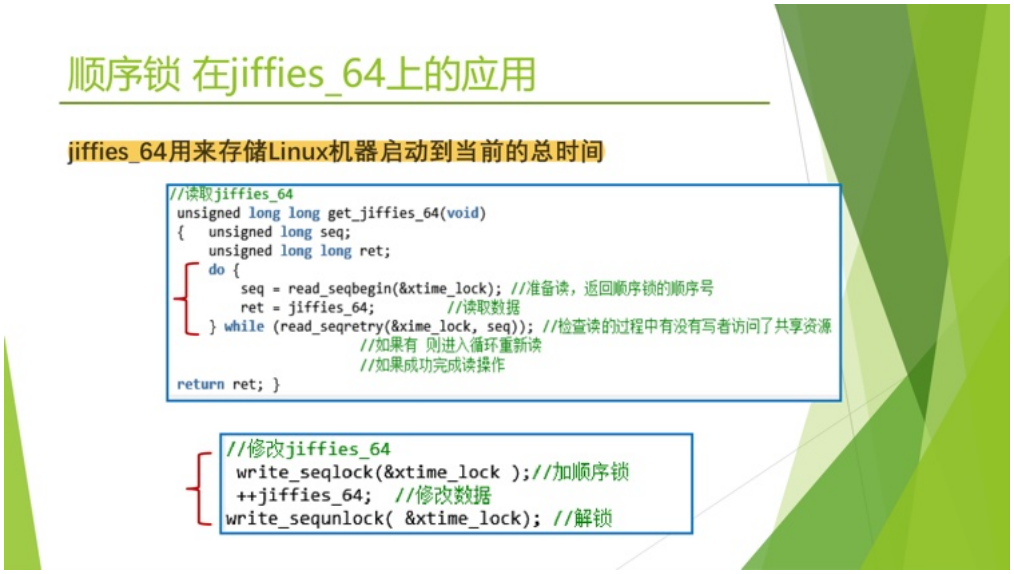
RCU 不能自动解决多个写者之间的互斥，写者仍需要锁。

10.7.6 完成变量



图示：完成变量用于一个执行单元等待另一个执行单元完成特定事件。

- Completion 表达“一次性事件完成”。
- 等待者调用 `wait_for_completion`。
- 完成者调用 `complete` 或 `complete_all`。
- 适合线程创建、退出、设备初始化、`vfork` 等场景。



图示：屏障让多个线程在某个点集合，全部到达后再一起继续。

10.7.7 屏障

> **思考题：**如何用 P/V 操作实现屏障？

解答：对一次性屏障，设共享计数 `count = 0`、互斥信号量 `mutex = 1`、门闩信号量 `turnstile = 0`、线程数 `N`：

```
barrier() {
    P(mutex);
    count++;
    if (count == N) {
        for (int i = 0; i < N; i++)
            V(turnstile);
    }
    V(mutex);

    P(turnstile);
}
```

每个线程到达后增加计数；最后一个到达者释放 `N` 次门闩，让所有线程通过。可复用屏障还需要第二道门闩，防止下一轮与本轮混淆。

10.8 本讲小结

- IPC 解决进程之间的信息交换，常与同步机制配合使用。
- 消息传递隔离性好，但有复制和内核队列开销。
- 共享内存速度快，但必须额外同步。
- 管道和套接字把通信抽象成字节流或端点。

- RPC 把网络通信封装成远程调用，但不能忽略分布式系统错误语义。
- Linux 内核同步机制根据上下文、临界区长度、读写比例和是否可睡眠选择不同工具。

LECTURE 11

第 11 讲

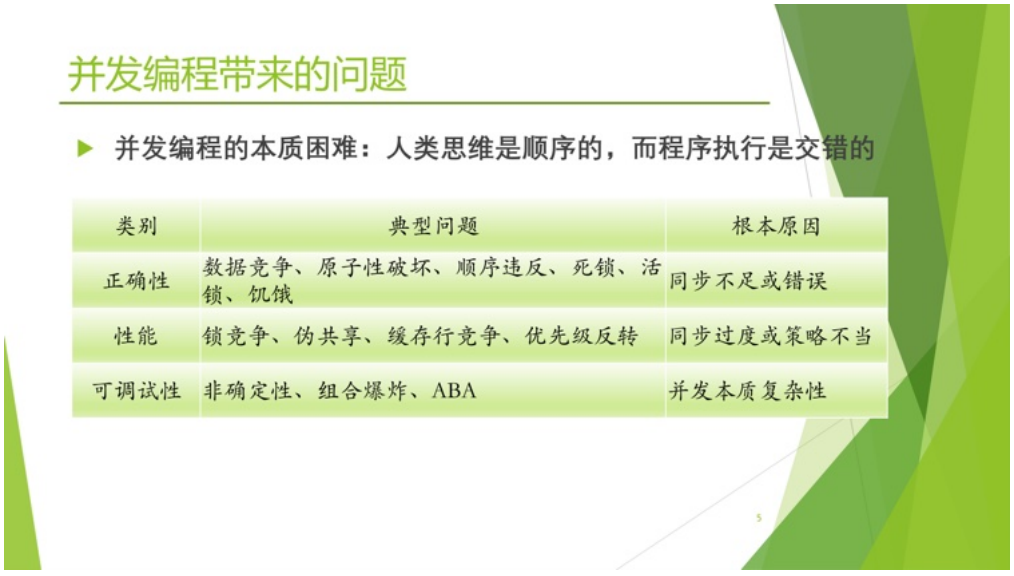
死锁

Operating Systems Course Notes

11.1 本讲导读

- 死锁是并发程序中最典型、也最难调试的问题之一。
- 本讲主线：
  1. 死锁、活锁、饥饿；
  2. 死锁产生的四个必要条件；
  3. 资源分配图与图化简；
  4. 死锁预防、避免、检测与解除；
  5. 安全状态与银行家算法；
  6. 哲学家就餐问题；
  7. 现代并发缺陷、信号处理死锁、思考题。

11.2 基本概念



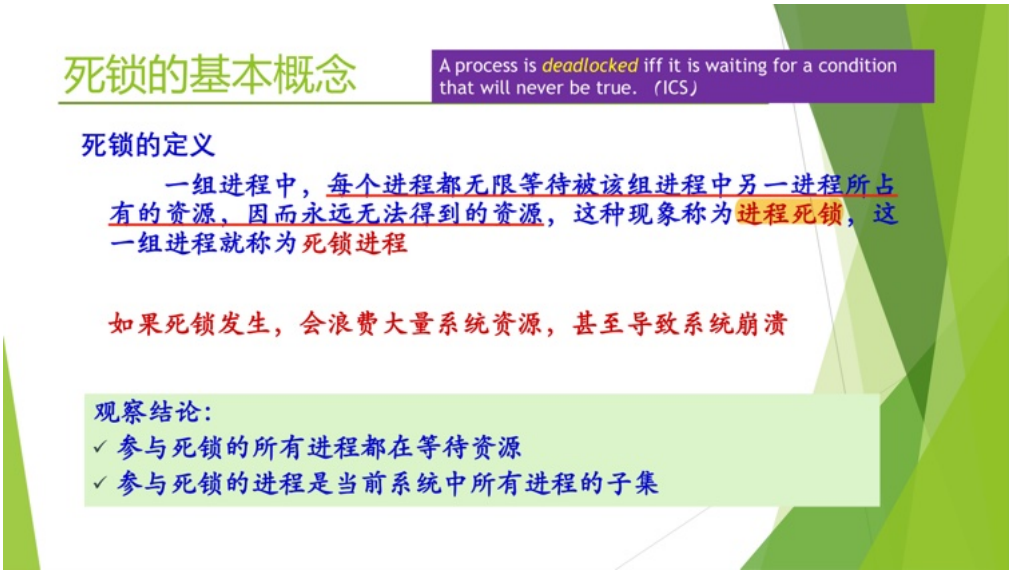
图示：并发程序会遇到正确性、性能和可调试性问题。

11.2.1 死锁

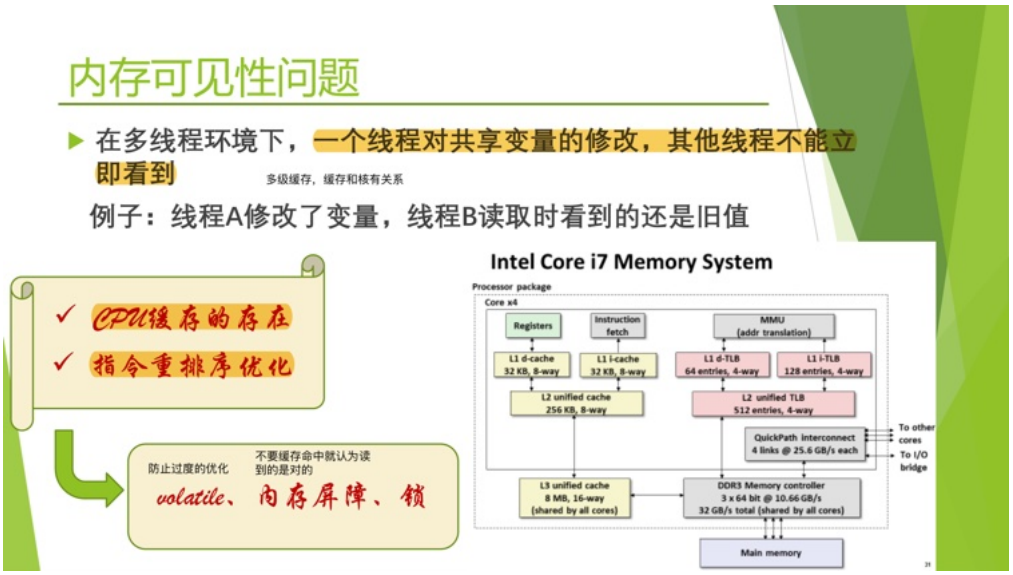
死锁（Deadlock）：一组进程中，每个进程都无限等待被该组中另一个进程占有的资源，导致所有进程都无法继续。

关键结论：

- 参与死锁的进程都处于等待状态；
- 死锁进程只是系统全部进程的一个子集；
- 死锁常由资源数量有限、锁顺序错误、封装隐藏锁依赖导致。



图示：一组进程彼此等待对方占有的资源，永远无法继续。

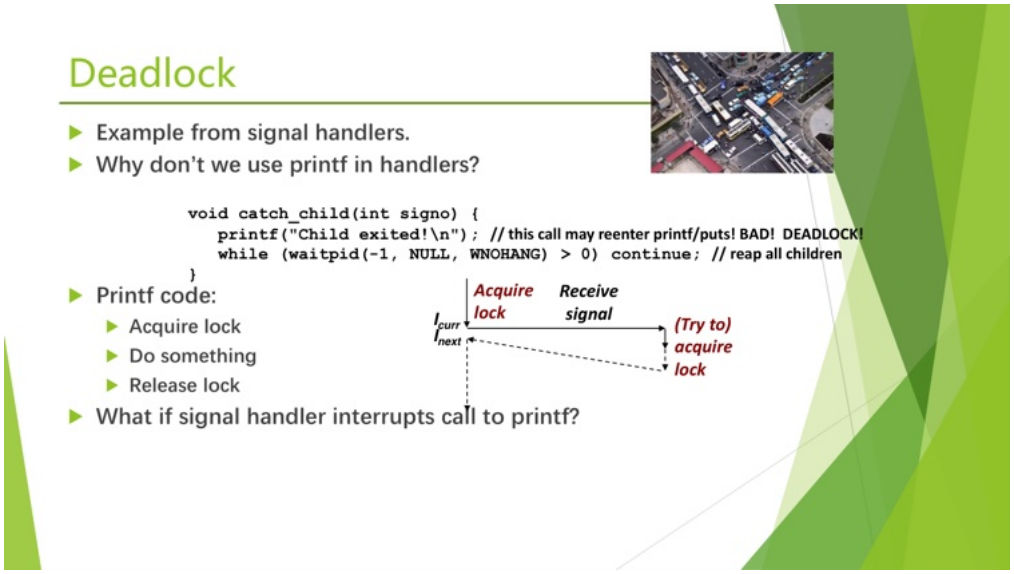


图示：并发问题不仅包括死锁，还包括内存可见性、原子性破坏和顺序错误。

11.2.2 活锁与饥饿

- **活锁（Livelock）**：进程没有阻塞，状态不断变化，但没有有效进展。
- **饥饿（Starvation）**：由于资源分配策略不公平，一个进程长期得不到资源。

问题	状态	是否有系统进展	典型原因
死锁	阻塞等待	无	循环等待资源
活锁	活跃运行	无有效进展	反复礼让/重试
饥饿	阻塞或就绪	其他进程可能有进展	不公平调度或分配



图示：主程序持有 `printf` 内部锁时被信号打断，handler 再调用 `printf` 可能死锁。

11.2.3 信号处理中的死锁

> **思考题：**为什么不应该在 signal handler 中调用 `printf`？

解答： `printf` 内部可能持有锁。若主程序正在执行 `printf` 并持有该锁，此时信号到达，handler 又调用 `printf`，handler 会试图获取同一把锁；主程序必须等 handler 返回才能释放锁，handler 又等主程序释放锁，于是死锁。因此 signal handler 应只调用 async-signal-safe 函数。

11.3 死锁产生的必要条件



图示：互斥使用、占有且等待、不可抢占、循环等待必须同时成立才可能死锁。

1. **互斥使用**：资源一次只能被一个进程使用。
2. **占有且等待**：进程已占有资源，同时等待新资源。

3. **不可抢占**：资源不能被系统强制夺走，只能由占有者释放。
4. **循环等待**：存在进程等待环路。

只要破坏任意一个必要条件，就能预防死锁。

11.4 资源分配图



图示：资源分配图用进程节点、资源节点、申请边和分配边描述系统状态。

资源分配图 $G = (V, E)$：

- 进程节点：$P_1, P_2, \ldots$；
- 资源节点：$R_1, R_2, \ldots$，方框内圆点表示实例数；
- 申请边：$P_i \to R_j$；
- 分配边：$R_j \to P_i$。

死锁定理：

- 无环：一定没有死锁；
- 有环：可能死锁；
- 若每类资源只有一个实例，有环是死锁的充要条件。

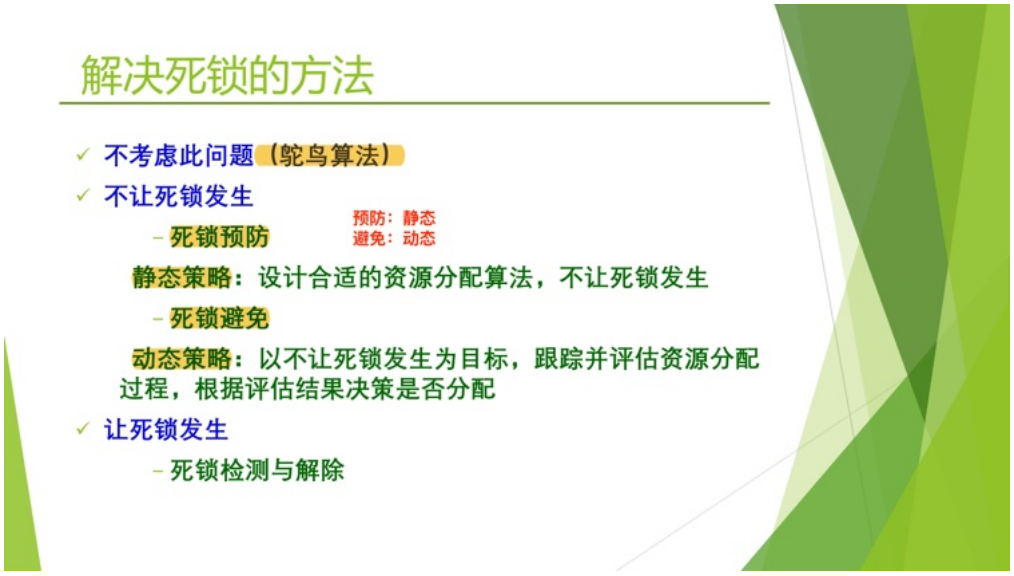
11.4.1 图化简

图化简步骤：

1. 找到一个不阻塞的进程；
2. 假设它运行完成并释放所有资源；
3. 用释放资源满足其他等待进程；
4. 重复直到不能化简；

5. 若所有进程都能化简，系统无死锁；否则剩余进程死锁。

11.5 死锁处理策略



图示：死锁处理包括忽略、预防、避免、检测与解除。

11.5.1 鸵鸟算法

- 假设死锁很少发生，交给应用或管理员处理。
- UNIX/Windows 常对一般资源死锁采用这种态度。
- 适合死锁概率低、检测处理成本高的场景。

11.5.2 死锁预防

破坏互斥

- 用资源转换技术把独占资源变为共享服务。
- 典型例子是 SPOOLing：进程不直接占用打印机，而是把打印任务提交给守护进程。

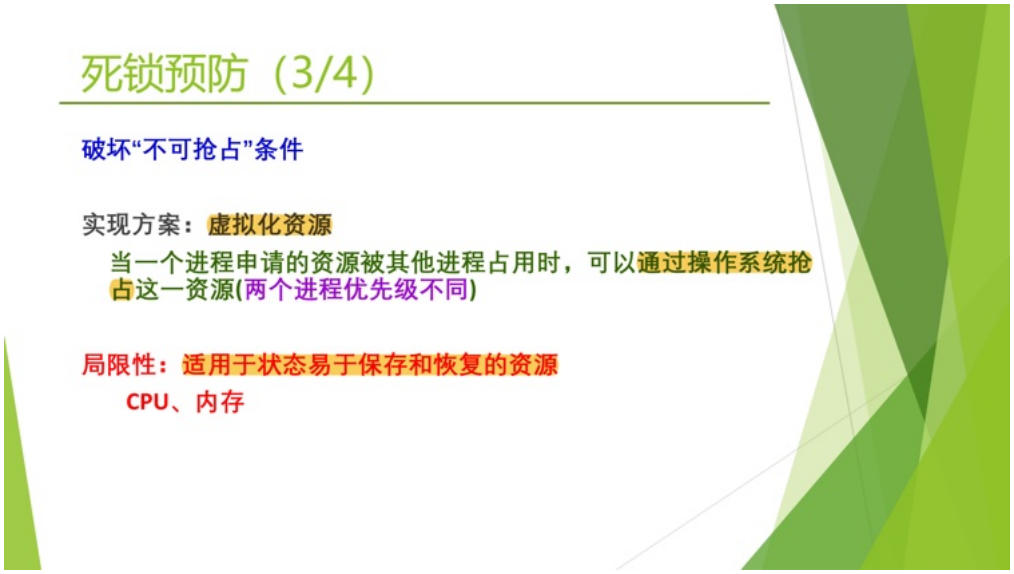
破坏占有且等待

- 一次性申请所有资源；
- 或申请新资源失败时释放已有资源。

缺点是资源利用率低、可能饥饿、程序设计不方便。

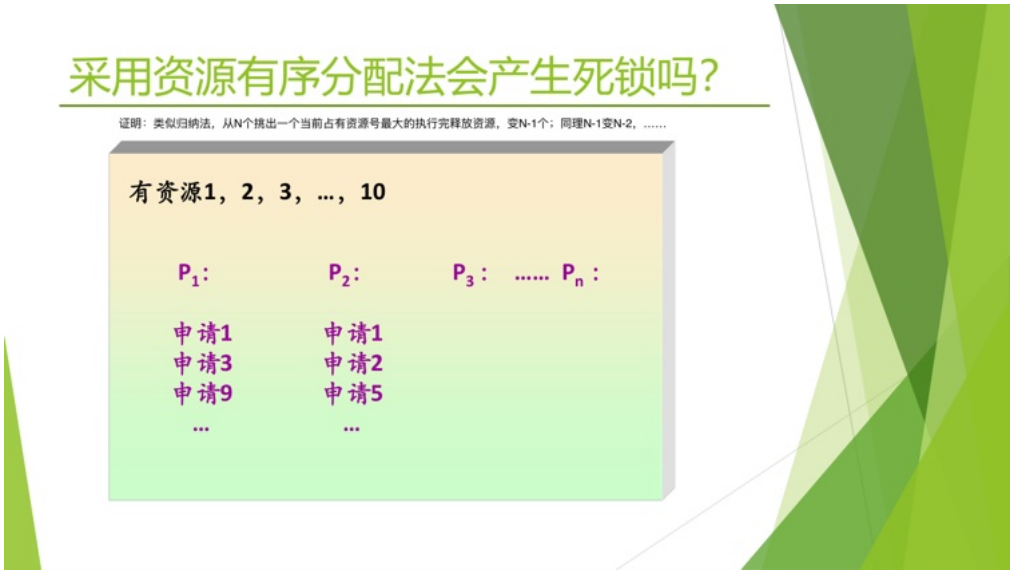
破坏不可抢占

- 允许系统抢占资源。
- 适合 CPU、内存等状态可保存资源。
- 不适合打印机、文件锁等难以回滚资源。



图示：SPOOLing 把独占打印机转换为共享打印服务。

破坏循环等待



图示：为资源编号，进程按递增顺序申请，避免等待环。

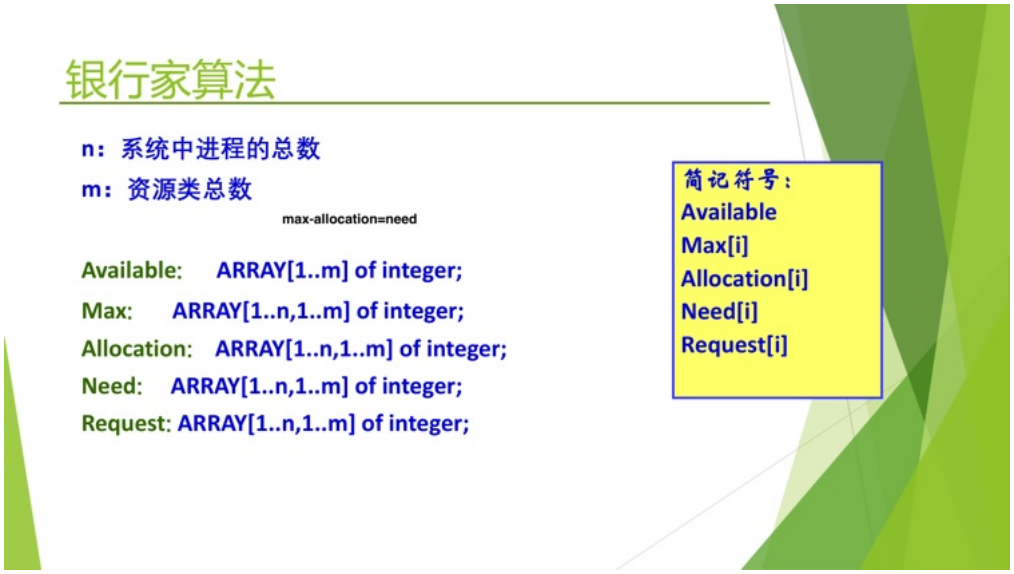
> **思考题：**采用资源有序分配法会产生死锁吗?

解答：不会。若所有进程只能按资源编号递增申请，则等待边总是从低编号资源持有者指向更高编号资源，不可能形成回到低编号资源的环。形式化地说，循环等待要求编号沿环严格递增又回到起点，矛盾。

11.6 死锁避免与银行家算法

11.6.1 安全状态

- **安全状态**：存在一个进程序列，使得每个进程都能依次获得所需资源、完成并释放资源。



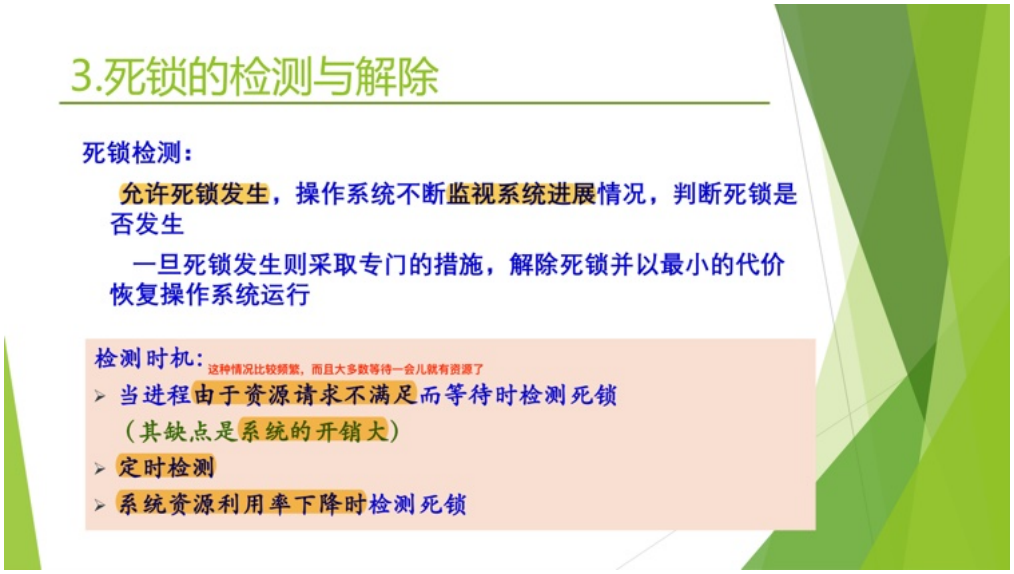
图示：安全状态存在一个安全序列；不安全状态不一定立即死锁，但可能走向死锁。

- **不安全状态**：不存在安全序列。
- 不安全状态不一定已经死锁，但可能导致死锁。

> **思考题：**不安全状态一定导致死锁吗？

解答：不一定。不安全只是表示“系统无法保证从当前状态一定可以让所有进程完成”。如果后续进程不继续申请最坏情况下的资源，仍可能运行完成。但操作系统不能依赖这种侥幸，因此死锁避免算法会拒绝进入不安全状态。

11.6.2 银行家算法



图示：银行家算法在分配前假设分配，然后检查系统是否仍处于安全状态。

核心数据结构：

- `Available[j]`：资源 j 当前可用数量；
- `Max[i][j]`：进程 i 最多需要资源 j 的数量；
- `Allocation[i][j]`：进程 i 已分配资源；
- `Need[i][j] = Max[i][j] - Allocation[i][j]`。

请求处理：

1. 检查 `Request_i <= Need_i`；
2. 检查 `Request_i <= Available`；
3. 假设分配；
4. 运行安全性算法；
5. 若安全则真正分配，否则让进程等待。

> **思考题：**银行家算法基本思想是什么？

解答：像银行放贷一样，系统不只看“现在有没有资源”，还看“分配后是否仍能保证所有客户最终完成”。只要一次分配会让系统进入不安全状态，即使当前资源足够，也暂时拒绝。

11.6.3 银行家算法题



图示：判断当前状态是否安全，并分析 P2、P5、P1 的请求能否分配。

> **思考题：**如何判断给定状态是否安全？

解答：使用安全性算法：

1. `Work = Available`，所有 `Finish[i] = false`；
2. 找一个 `Finish[i] == false` 且 `Need_i <= Work` 的进程；
3. 假设它完成，`Work += Allocation_i`，`Finish[i] = true`；

4. 重复直到找不到新进程；
5. 若全部 `Finish` 为真，则安全，找到的顺序就是安全序列。

对于题中三个请求，处理方法相同：先检查请求是否超过 Need 和 Available，再做“试分配 + 安全性检查”。能找到安全序列则可分配，否则不可分配。不要只因为当前 `Available` 足够就直接分配。

11.7 死锁检测与解除

- 检测：周期性运行资源分配图化简或等待图环检测。
- 解除：
  - 撤销所有死锁进程；
  - 逐个撤销进程直到死锁消失；
  - 抢占资源并回滚；
  - 人工干预。

选择牺牲进程时考虑：

- 进程优先级；
- 已运行时间；
- 已占资源数量；
- 还需资源数量；
- 回滚代价；
- 用户影响。

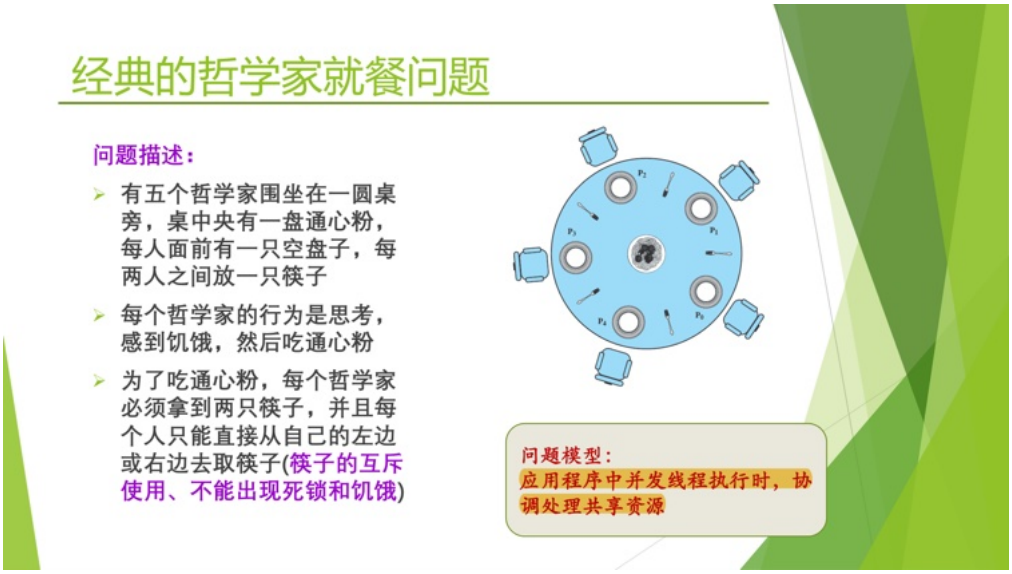
11.8 哲学家就餐问题

问题模型：

- 筷子是互斥资源；
- 每个哲学家需要左右两只筷子才能吃饭；
- 若所有哲学家都先拿左筷子，再等右筷子，会形成循环等待。

11.8.1 朴素方案

```
P(fork[i]);
P(fork[(i+1)%5]);
eat();
V(fork[(i+1)%5]);
V(fork[i]);
```

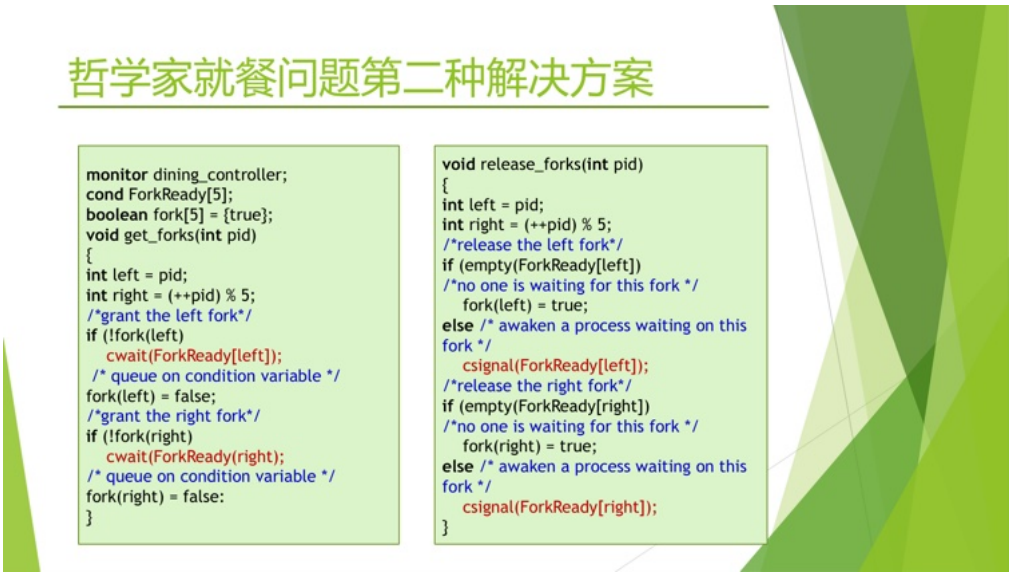


图示：五个哲学家共享五只筷子，必须协调互斥使用并避免死锁和饥饿。

> **思考题：**何时发生死锁？

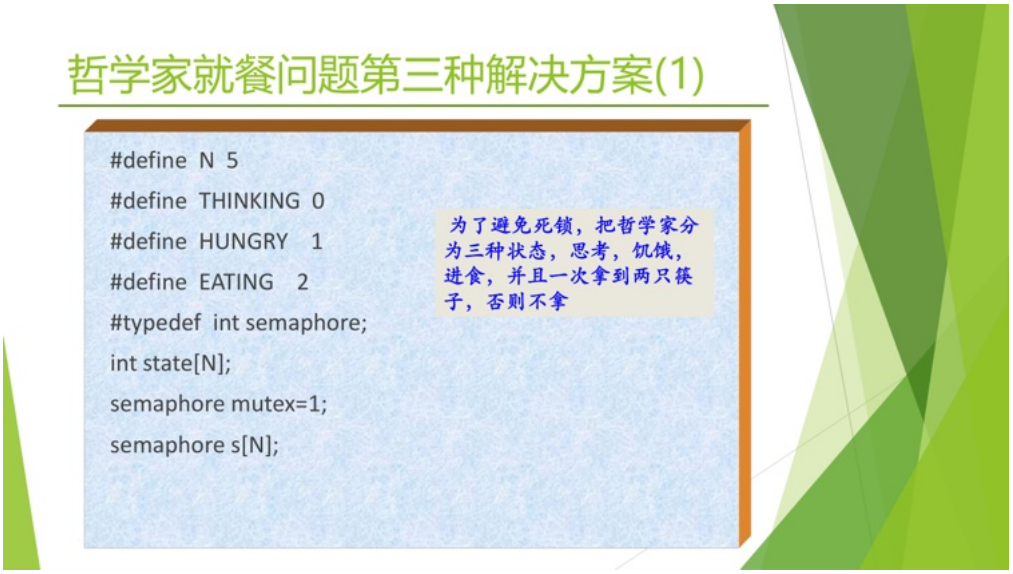
解答：当五个哲学家同时拿起自己左边的筷子后，每个人都等待右边筷子，而右边筷子被邻居占有，形成五进程循环等待。

11.8.2 管程方案



图示：管程统一管理哲学家状态和条件变量。

- 管程维护每个哲学家状态：THINKING、HUNGRY、EATING。
- 哲学家饿了后进入管程，只有左右邻居不在吃时才能吃。
- 吃完后测试左右邻居是否可以被唤醒。



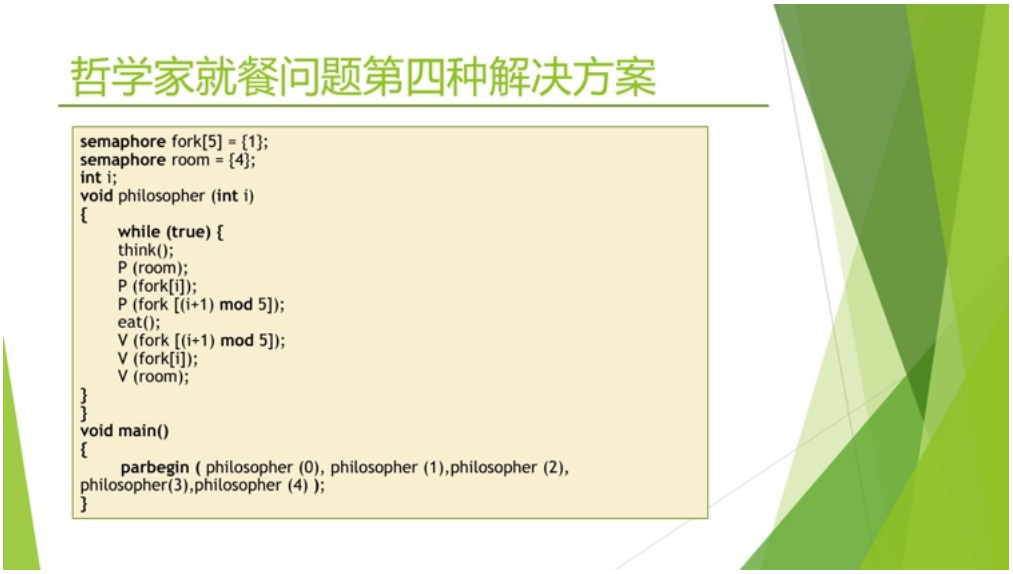
图示：一次拿到两只筷子，否则不拿，避免占有一只筷子等待另一只。

11.8.3 状态数组方案

核心思想：

- 不让哲学家先占一只筷子再等待另一只；
- 只有左右两只都可用时才进入吃饭状态；
- 每个哲学家有自己的信号量 `s[i]`；
- 全局 `mutex` 保护状态数组。

11.8.4 限制人数方案



图示：最多允许 4 个哲学家同时竞争筷子，从而破坏循环等待。

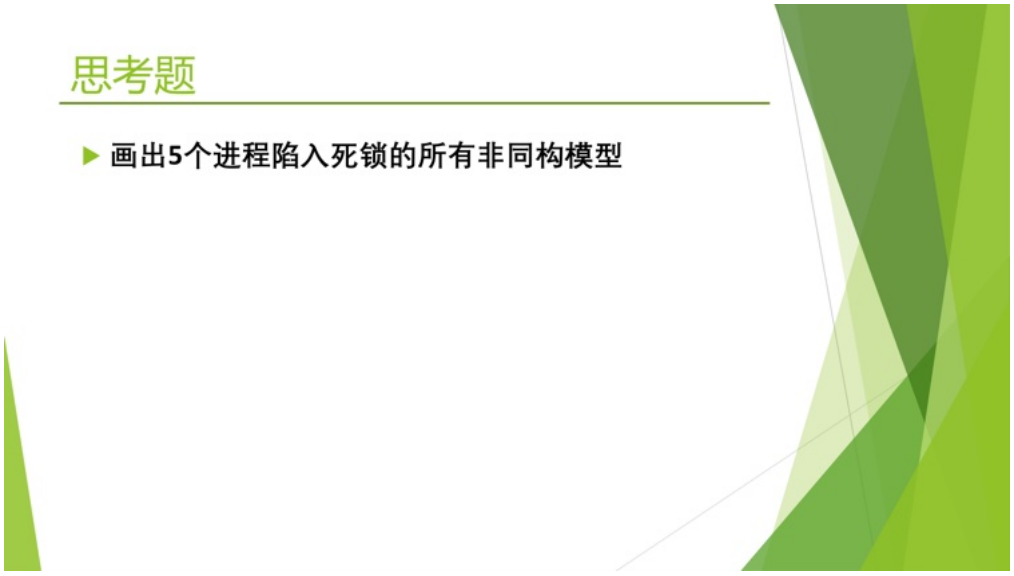
设 `room = 4`：

```
1 P(room);
```

```
P(fork[i]);
P(fork[(i+1)%5]);
eat();
V(fork[(i+1)%5]);
V(fork[i]);
V(room);
```

最多四人同时拿筷子，因此总有一个哲学家的两边筷子至少不会都被“形成闭环”的方式占住，可打破循环等待。

11.9 思考题整理



图示：思考 5 个进程陷入死锁的所有非同构等待模型。

> **思考题：**画出 5 个进程陷入死锁的所有非同构模型。

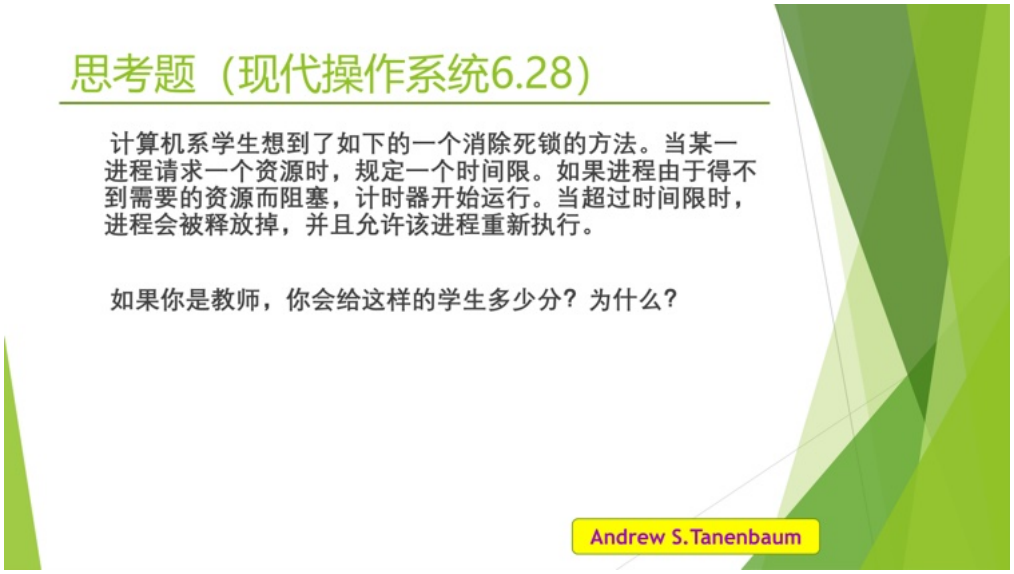
解答：若抽象为等待图，死锁要求参与进程构成至少一个有向环，且每个死锁进程都在环或强连通等待结构中。非同构模型可按强连通有向图分类：

- 最小 5 环：$P_1 \to P_2 \to P_3 \to P_4 \to P_5 \to P_1$；
- 5 环加一条弦：在基本环上增加额外等待边；
- 多个共享等待目标但整体仍强连通；
- 完全更密集的强连通等待图。

考试中重点不是枚举所有图论同构类型，而是说明“死锁等待图必须包含一个覆盖相关进程的循环等待结构”。若每个进程都只等待一个资源/进程，则非同构核心就是一个 5 环。

> **思考题：**学生提出：进程请求资源超过时间限就释放掉并重新执行。你会给多少分？为什么？

解答：可以给部分分，但不能给满分。



图示：用超时释放进程并重新执行来消除死锁的思考题。

优点：

- 能打破长期等待；
- 类似某种“检测 + 恢复”策略；
- 实现思路直观。

问题：

- 超时不等于死锁，可能误杀只是等待较慢的进程；
- 释放进程可能破坏外部状态，需要回滚机制；
- 进程重新执行可能再次死锁或活锁；
- 难以选择合理时间限；
- 对不可抢占资源、I/O 副作用、事务一致性处理不足。

因此它是一种粗糙的恢复策略，不是严格的死锁预防或避免算法。

11.10 本讲小结

- 死锁需要四个必要条件同时成立。
- 资源分配图帮助判断等待关系；单实例资源中有环即死锁。
- 预防通过破坏必要条件，避免通过保持安全状态，检测与解除允许死锁发生后处理。
- 银行家算法的核心是“试分配后仍安全才分配”。
- 哲学家就餐问题浓缩了互斥、循环等待、饥饿和公平性。

LECTURE 12

第 12 讲

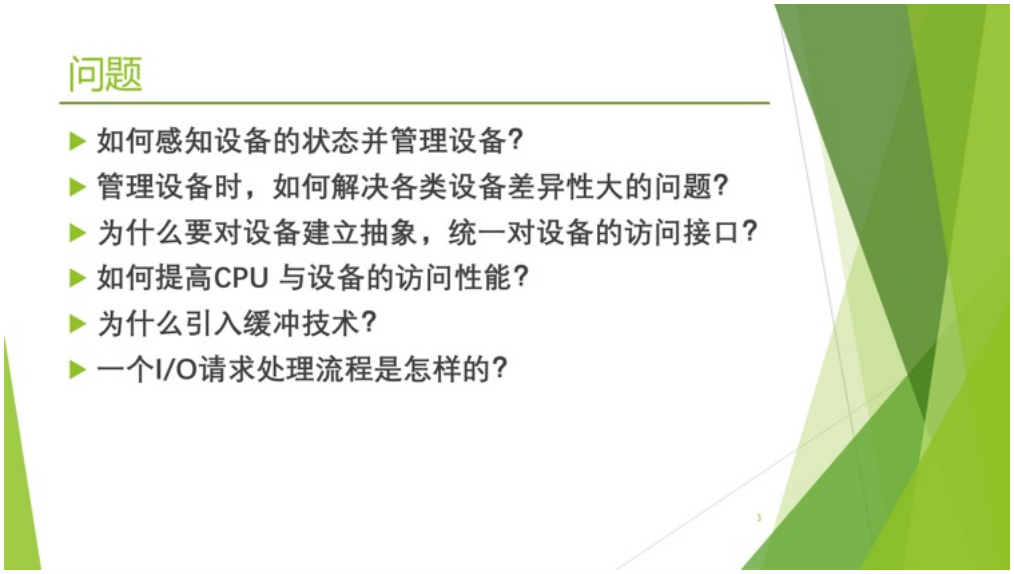
I/O 管理

Operating Systems Course Notes

12.1 本讲导读

- I/O 管理连接应用、文件系统、设备驱动和硬件，是操作系统复杂性的主要来源之一。
- 本讲主线：
  1. I/O 的特点和设备发展；
  2. 设备分类和 I/O 系统结构；
  3. 设备控制器、寄存器、端口 I/O 与内存映射 I/O；
  4. 程序查询、中断驱动、DMA；
  5. I/O 软件分层、设备独立性；
  6. 缓冲技术、设备分配、驱动程序、I/O 进程；
  7. I/O 性能问题与异步 I/O。

12.2 开篇问题



图示：设备状态感知、设备差异抽象、统一接口、CPU 与设备性能、缓冲技术和 I/O 请求流程。

> **思考题：**如何感知设备状态并管理设备?

解答：操作系统通过设备控制器提供的状态寄存器、控制寄存器、数据寄存器或内存映射寄存器感知设备状态，并通过设备驱动程序解释状态位、发送命令、处理中断、管理请求队列。

> **思考题：**管理设备时如何解决各类设备差异大的问题?

解答：通过分层和抽象。上层使用统一的文件或 I/O 接口；设备无关层负责命名、保护、缓冲和分配；设备驱动封装具体硬件寄存器和协议；中断处理程序处理设备完成事件。这样硬件差异被限制在驱动层。

> **思考题：**为什么要对设备建立抽象，统一访问接口？

解答：统一接口降低应用复杂度，提高可移植性，并允许重定向、缓冲、权限检查、调度和虚拟设备等机制在统一框架中实现。UNIX 把设备作为特殊文件就是典型例子。

> **思考题：**如何提高 CPU 与设备的访问性能？

解答：主要靠中断、DMA、缓冲、异步 I/O、请求合并、设备队列调度和并行化。目标是让 CPU 不被低速 I/O 长时间阻塞，并尽量让 CPU 与设备重叠运行。

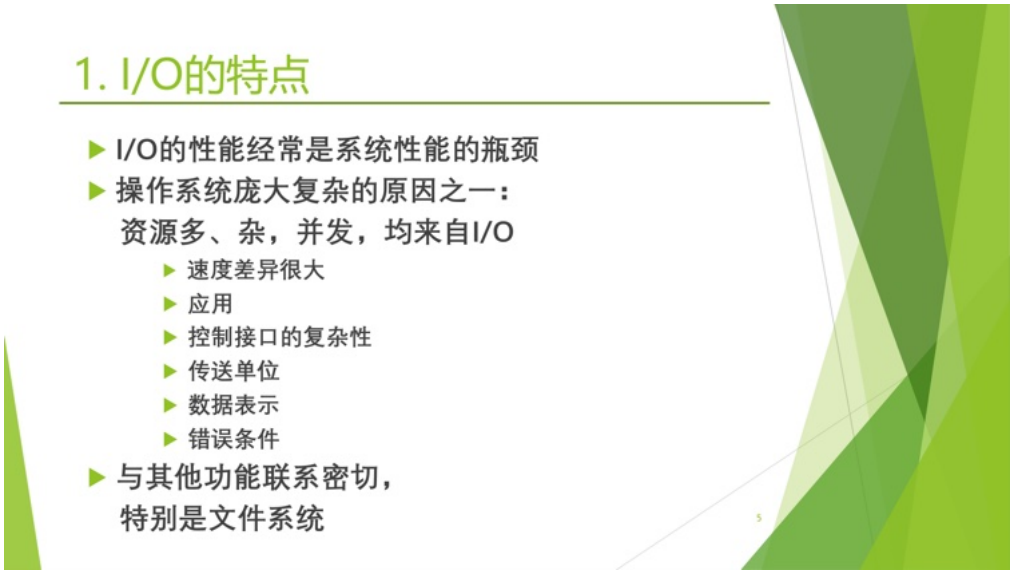
> **思考题：**为什么引入缓冲技术？

解答：缓冲解决生产者和消费者速度不匹配。CPU、内存、磁盘、网络、终端速度差异巨大，缓冲可以平滑突发、批量传输、减少中断次数，并支持异步化。

> **思考题：**一个 I/O 请求处理流程是怎样的？

解答：应用发起系统调用；文件系统或设备无关层检查参数和权限；驱动程序构造设备命令；控制器执行 I/O；完成后产生中断；中断处理程序唤醒等待进程；内核把结果返回用户态。

12.3 I/O 管理概述



图示：I/O 性能常是系统瓶颈，设备种类多、速度差异大、控制接口复杂。

12.3.1 I/O 的特点

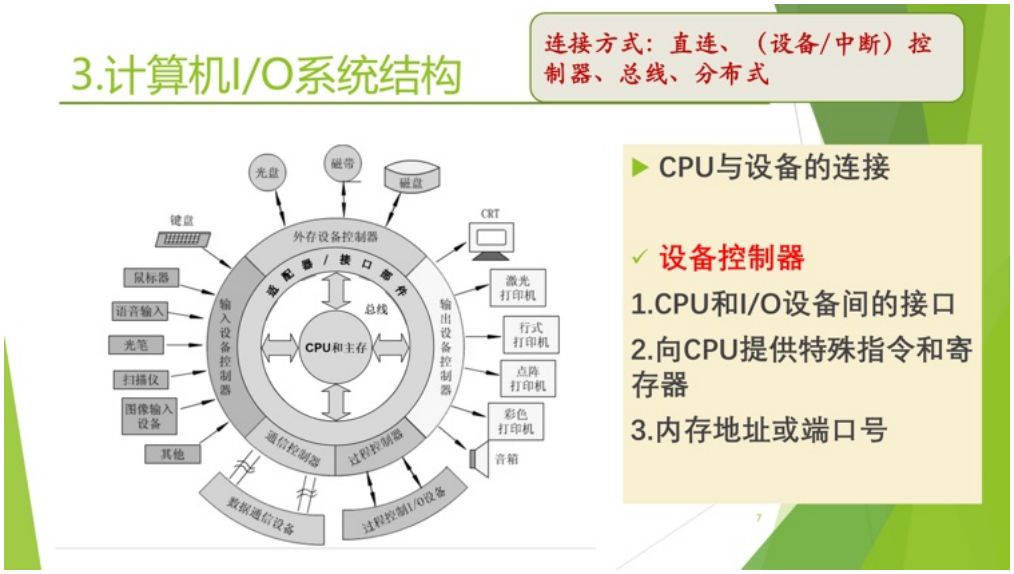
- I/O 经常是系统性能瓶颈；
- 设备种类多、数量多、接口复杂；

- 速度差异极大；
- 传送单位不同：字节、块、包；
- 错误条件复杂；
- 与文件系统、内存管理、中断、调度联系密切。

12.3.2 设备发展

- 简单设备：CPU 直接通过接口控制；
- 多设备系统：引入控制器和总线；
- 支持中断：减少 CPU 轮询；
- 支持 DMA：减少 CPU 参与数据搬运；
- 新型设备：GPU、声卡、智能网卡、RDMA 等。

12.4 I/O 系统结构与设备分类



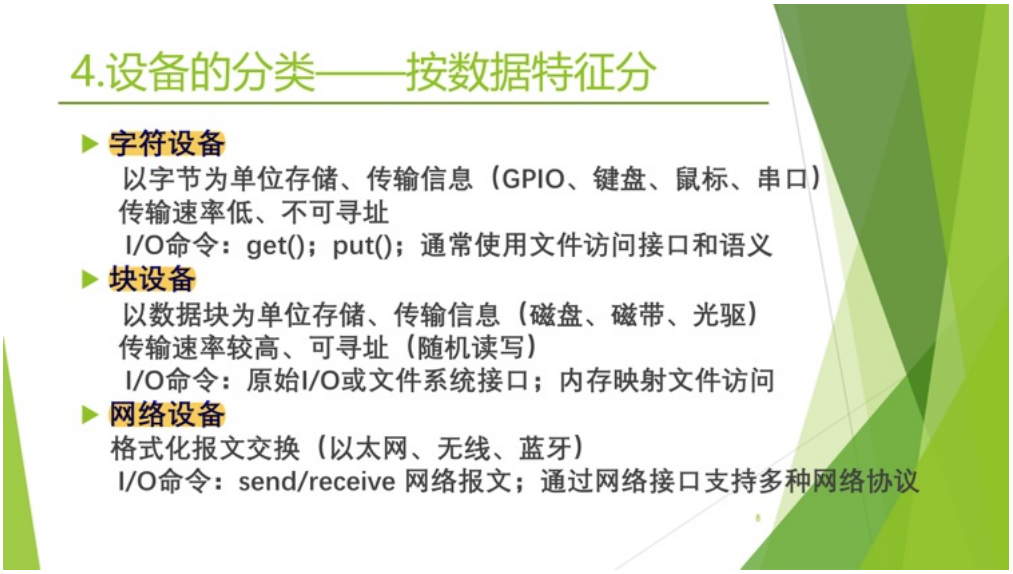
图示：CPU 通过设备控制器、总线和中断控制器连接 I/O 设备。

12.4.1 设备控制器

设备控制器是 CPU 和设备之间的接口：

- 接收 CPU 命令；
- 暴露控制寄存器、状态寄存器、数据寄存器；
- 完成设备内部协议；
- 产生中断；
- 支持缓冲、校验、DMA 等功能。

12.4.2 按数据特征分类



图示：字符设备、块设备、网络设备有不同传输单位和访问语义。

类型	传输单位	特点	例子
字符设备	字节流	低速、不可寻址	键盘、鼠标、串口
块设备	数据块	可寻址、随机访问	磁盘、光驱
网络设备	报文	协议驱动、包交换	以太网、无线网卡

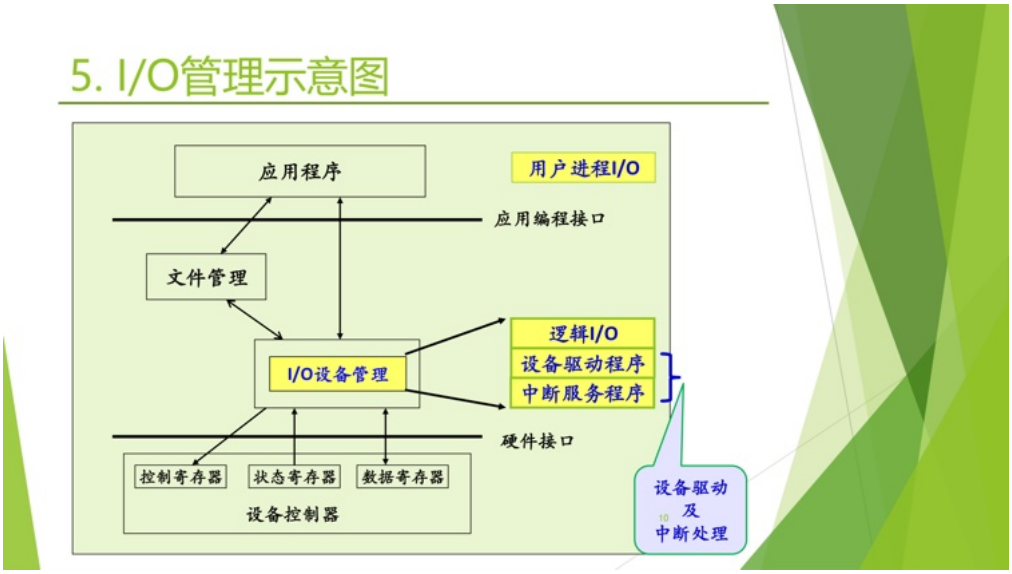
12.4.3 按资源分配分类

- **独占设备**：一段时间只能一个进程使用，如打印机、磁带。
- **共享设备**：多个进程交叉使用，如硬盘。
- **虚设备**：在共享设备上模拟独占设备，如 SPOOLing 用磁盘模拟打印队列。

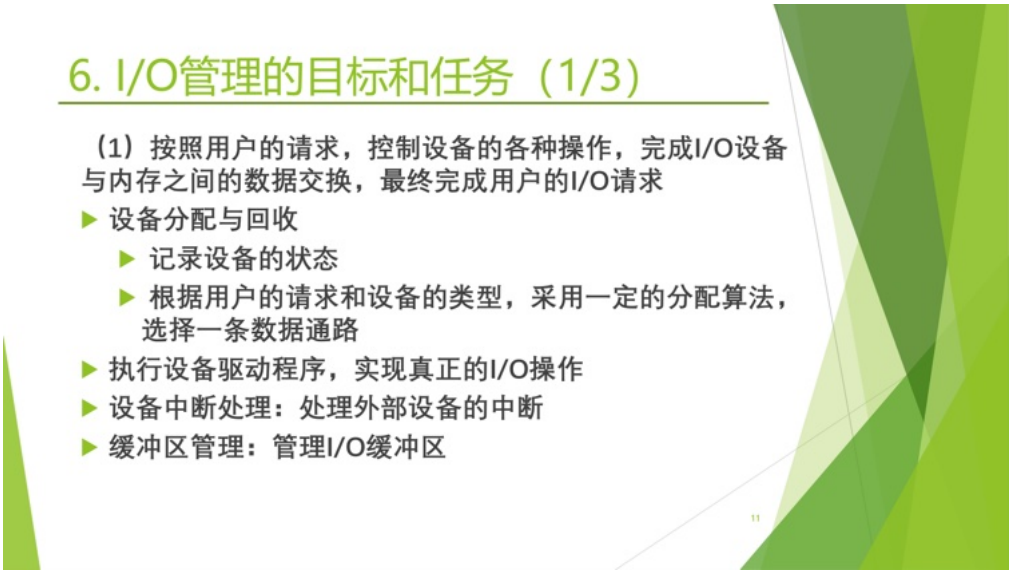
12.5 I/O 管理目标和任务

目标：

1. 按用户请求控制设备操作；
2. 完成设备与内存之间的数据交换；
3. 记录设备状态；
4. 分配和回收设备；
5. 选择数据通路；
6. 执行驱动程序；
7. 处理中断；
8. 管理缓冲区；
9. 建立统一、设备无关的接口；



图示：应用、文件管理、I/O 设备管理、驱动程序、中断处理和硬件接口之间的关系。



图示：设备管理要控制设备操作、完成设备与内存之间的数据交换。

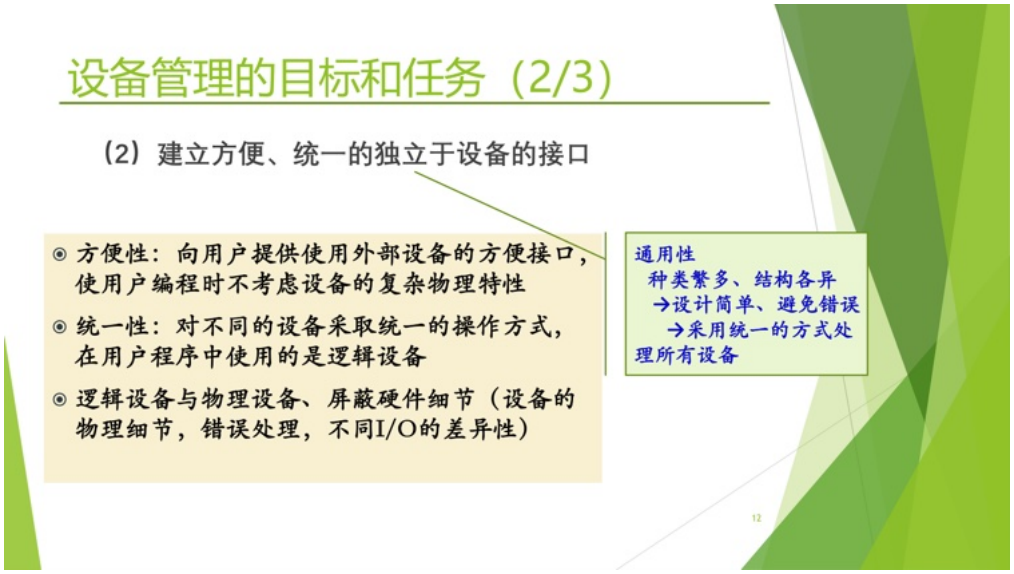
10. 提高 CPU 与设备并行度；
11. 保护数据安全。

设备独立性：

- 用户使用逻辑设备名；
- 系统负责逻辑设备到物理设备映射；
- 程序不依赖具体硬件；
- 方便 I/O 重定向和设备替换。

12.6 设备驱动模型

- 应用看到统一 API；



图示：逻辑设备与物理设备分离，屏蔽硬件细节。



图示：通过中间层和函数指针表封装底层设备差异，向上提供统一接口。

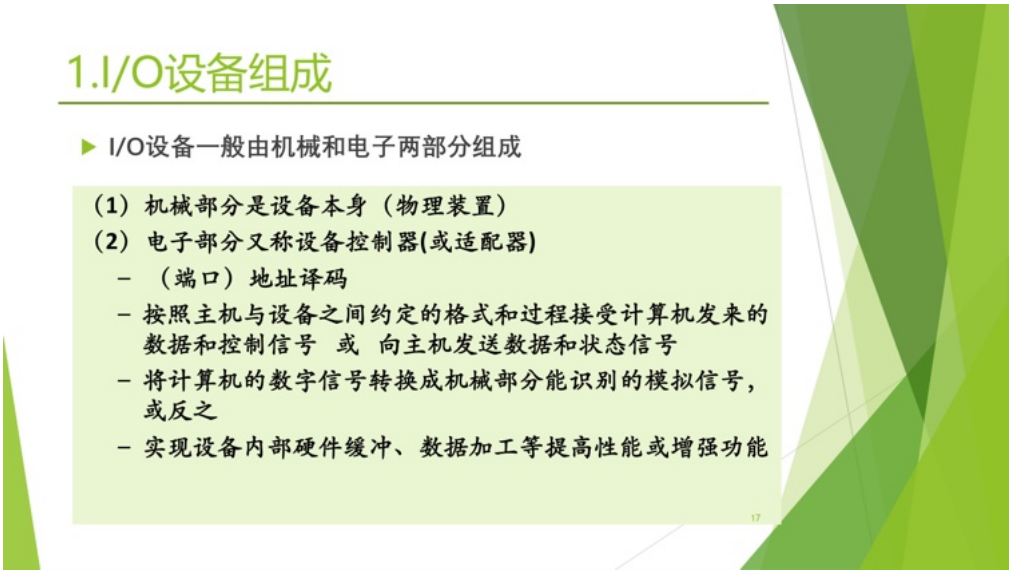
- VFS/设备无关层处理通用逻辑；
- 驱动程序实现具体设备操作；
- 函数指针表提供类似多态的机制；
- `struct file`、`struct inode`、`struct dentry` 等对象把设备纳入文件系统框架。

12.7 I/O 硬件组成

I/O 设备通常包含：

- 机械部分：实际执行物理动作；
- 电子部分：设备控制器或适配器。

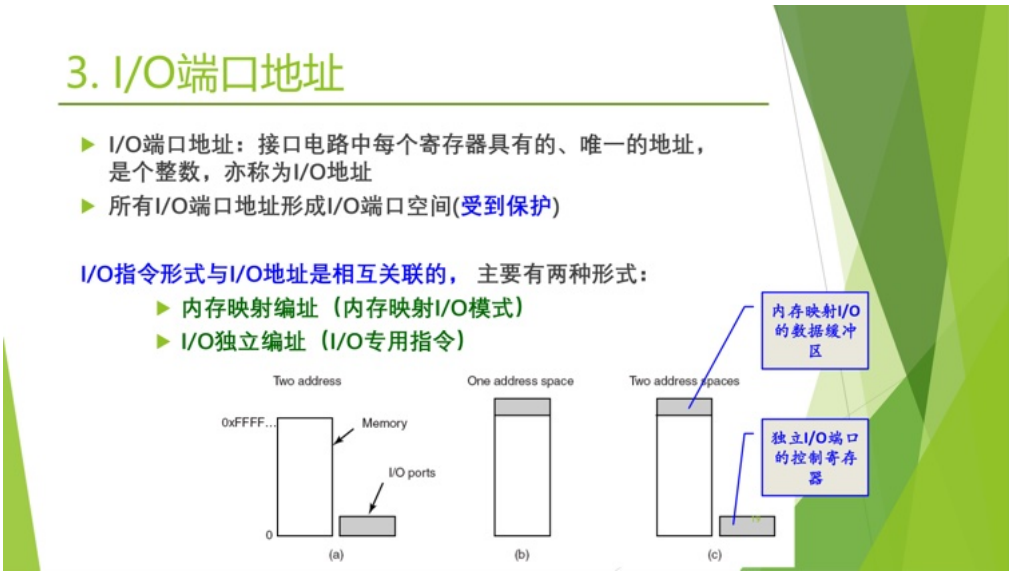
控制器职责：



图示：I/O 设备由机械部分和电子控制器组成。

- 地址译码；
- 接收控制信号和数据；
- 返回状态和结果；
- 数字信号与设备物理信号转换；
- 内部缓冲和数据处理。

12.7.1 控制器接口



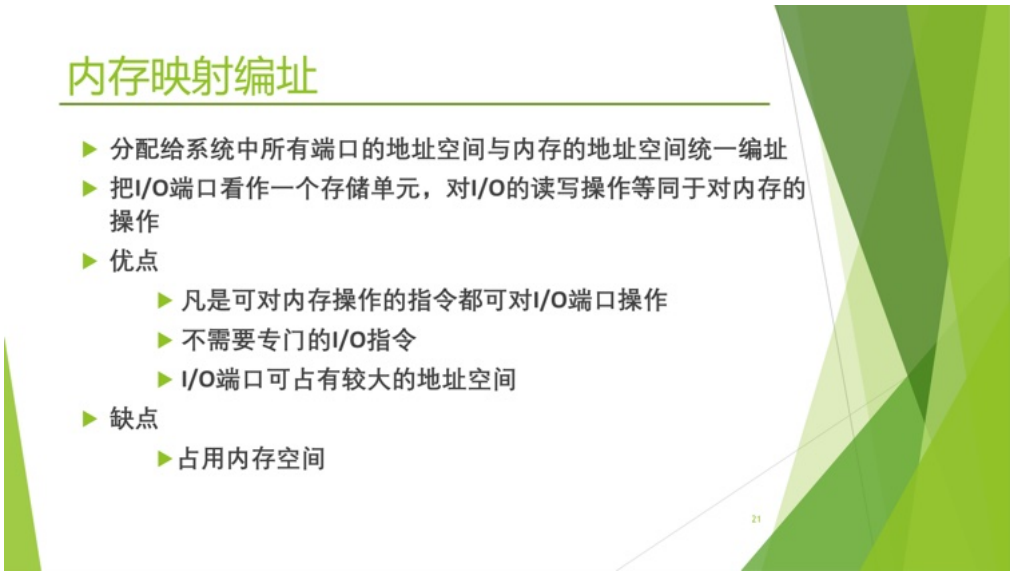
图示：操作系统通过控制器接口寄存器写命令、读状态和传输数据。

常见寄存器：

- 控制寄存器：写入命令；
- 状态寄存器：读取设备状态；

- 数据寄存器：传输数据；
- 地址/长度寄存器：DMA 或块传输参数。

12.7.2 I/O 地址空间



图示：设备寄存器可通过专用 I/O 指令或内存地址访问。

两种方式：

- **端口 I/O**：使用特殊 I/O 指令访问端口号；
- **内存映射 I/O（MMIO）**：把设备寄存器映射到物理地址空间，用普通 load/store 访问。

> **思考题：**为什么设备控制寄存器不能被普通高速缓存缓存？

解答：设备寄存器反映外设实时状态。如果第一次读取状态寄存器后被缓存，后续循环可能一直读到旧值，看不到设备已经就绪。MMIO 区域必须标记为不可缓存或使用特殊内存属性，并配合内存屏障保证访问顺序。

12.8 I/O 控制方式

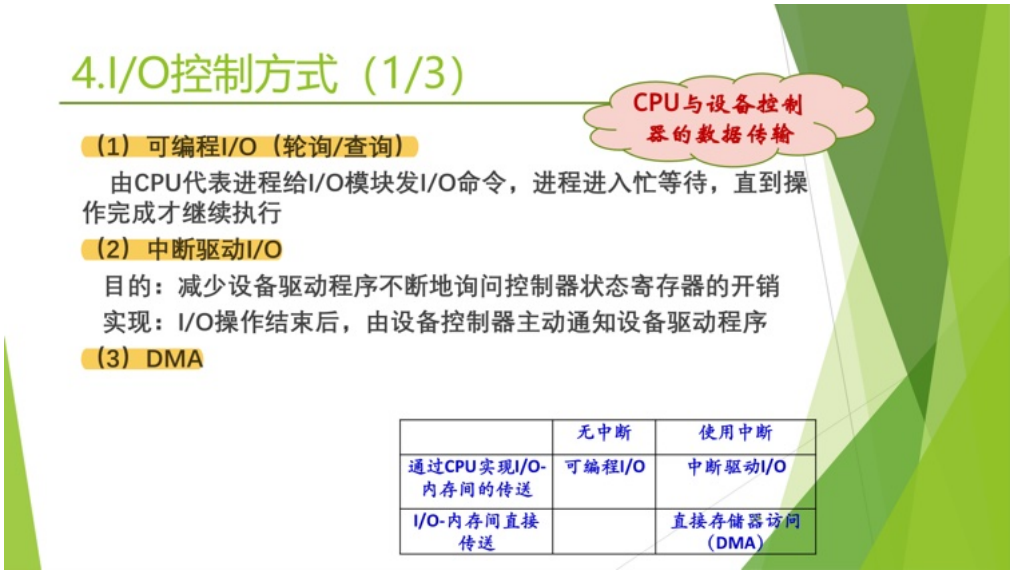
12.8.1 程序查询 I/O

CPU 不断轮询状态寄存器：

- 实现简单；
- 适合极简单或极快设备；
- 浪费 CPU。

12.8.2 中断驱动 I/O

- CPU 发出 I/O 命令后继续执行其他任务；



图示：中断驱动 I/O 由设备完成后通知 CPU，DMA 进一步减少 CPU 数据搬运。

- 设备完成后产生中断；
- 中断处理程序完成收尾并唤醒等待进程；
- 减少轮询开销。

12.8.3 DMA

- CPU 设置源地址、目标地址、长度、方向；
- DMA 控制器直接在设备和内存之间搬运数据；
- 完成后中断 CPU；
- 适合高速块设备和网卡；
- 需要处理缓存一致性和内存权限。

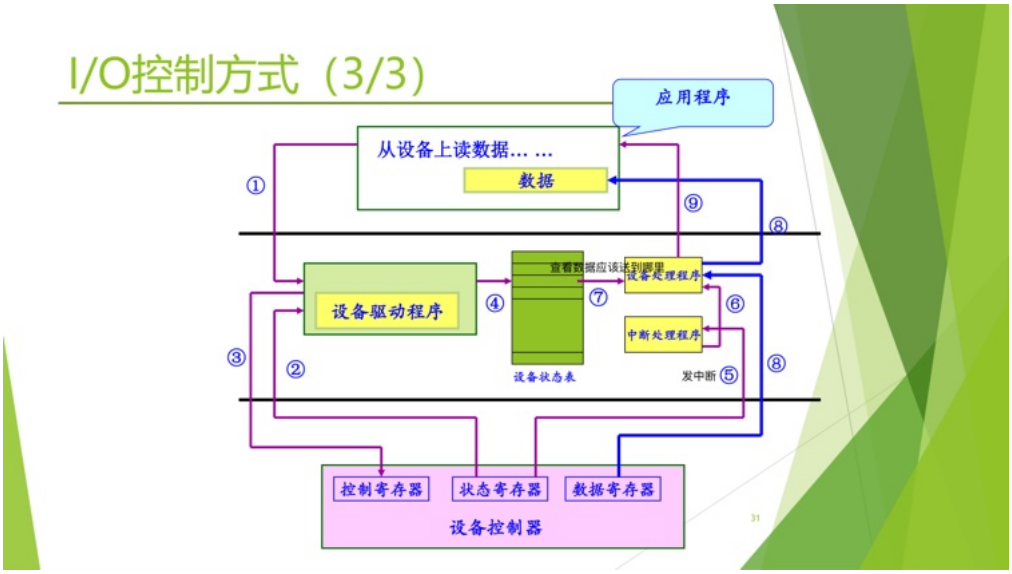
12.9 I/O 软件组成

典型层次：

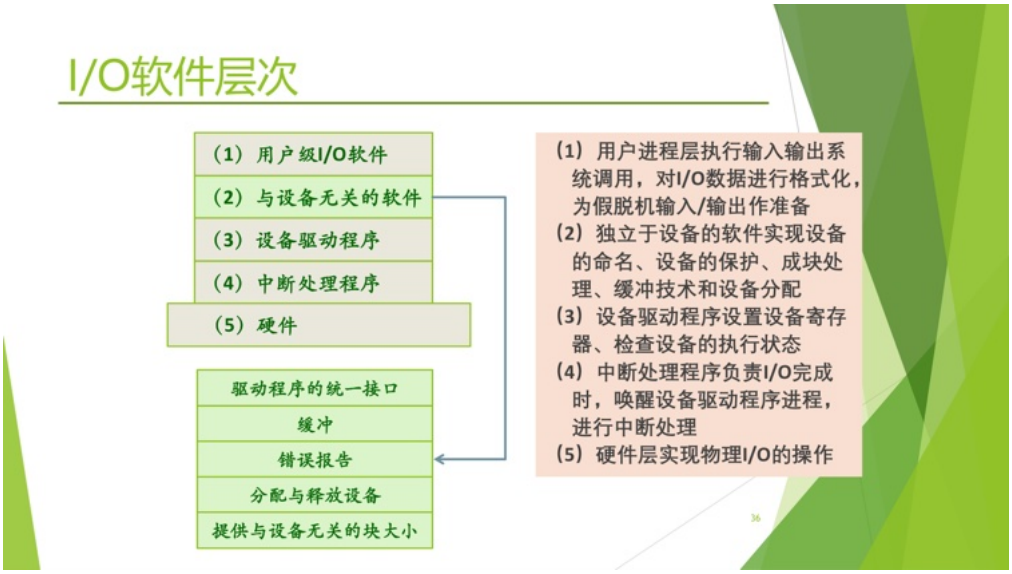
1. 用户级 I/O 软件：库函数、格式化、假脱机；
2. 设备无关软件：命名、保护、缓冲、分配、错误处理；
3. 设备驱动程序：具体设备命令与寄存器；
4. 中断处理程序：响应设备完成事件；
5. 硬件：执行真实 I/O。

12.9.1 设备命名和保护

- 逻辑设备名屏蔽物理设备；
- 设备作为特殊文件处理；



图示：I/O 软件包括用户级 I/O、设备无关软件、驱动程序、中断处理和硬件层。



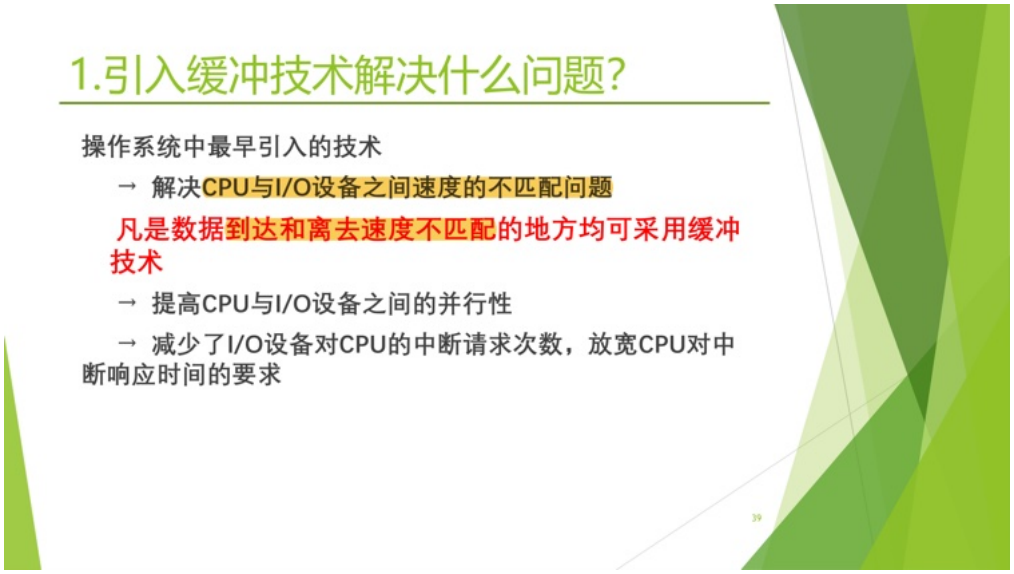
图示：设备独立性带来设备分配灵活性和 I/O 重定向。

- 文件系统完成权限检查和参数处理；
- 具体设备操作通过驱动程序函数表分派。

12.10 缓冲技术

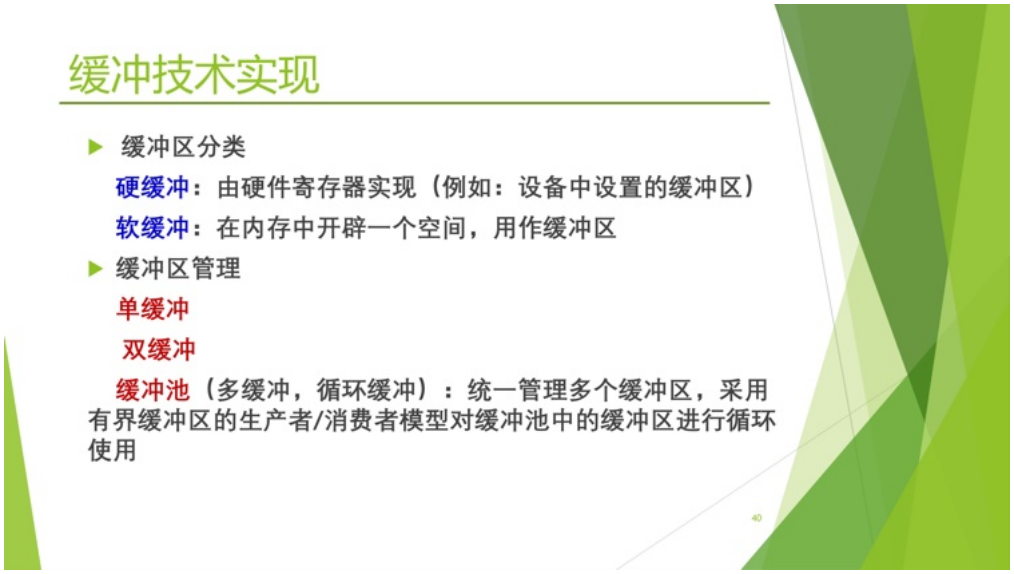
12.10.1 为什么需要缓冲

- 平滑速度差异；
- 减少中断次数；
- 支持批量传输；
- 支持生产者/消费者异步执行；
- 方便设备和应用之间格式转换。



图示：缓冲技术用于解决 CPU 与 I/O 设备速度不匹配。

12.10.2 缓冲分类

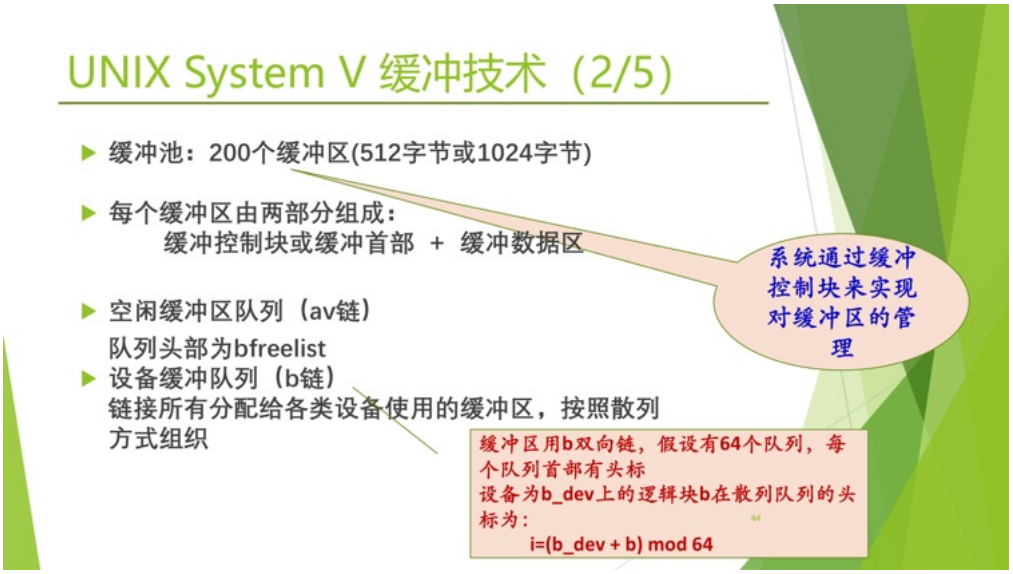


图示：硬缓冲由设备寄存器或控制器实现，软缓冲由内存实现。

- **硬缓冲**：设备或控制器内部缓冲区。
- **软缓冲**：操作系统在内存中开辟的缓冲区。
- **单缓冲**：一个缓冲区。
- **双缓冲**：一个缓冲区供设备填充，另一个供 CPU 处理。
- **循环缓冲**：多个缓冲区形成环形队列。
- **缓冲池**：系统统一管理多个缓冲区。

12.10.3 缓冲控制块

缓冲控制块通常包含：



图示：系统通过缓冲控制块管理缓冲区状态、队列和数据指针。

- 缓冲区地址；
- 缓冲区大小；
- 设备号或块号；
- 状态：空、满、忙、脏；
- 队列指针；
- 锁和等待队列。

12.11 设备分配

- 独占设备需要分配和释放；
- 静态分配：进程运行前分配，结束后回收，简单但利用率低；
- 动态分配：按需申请和释放，利用率高但要避免死锁；
- 分配算法要考虑公平、优先级、设备忙闲、数据通路。

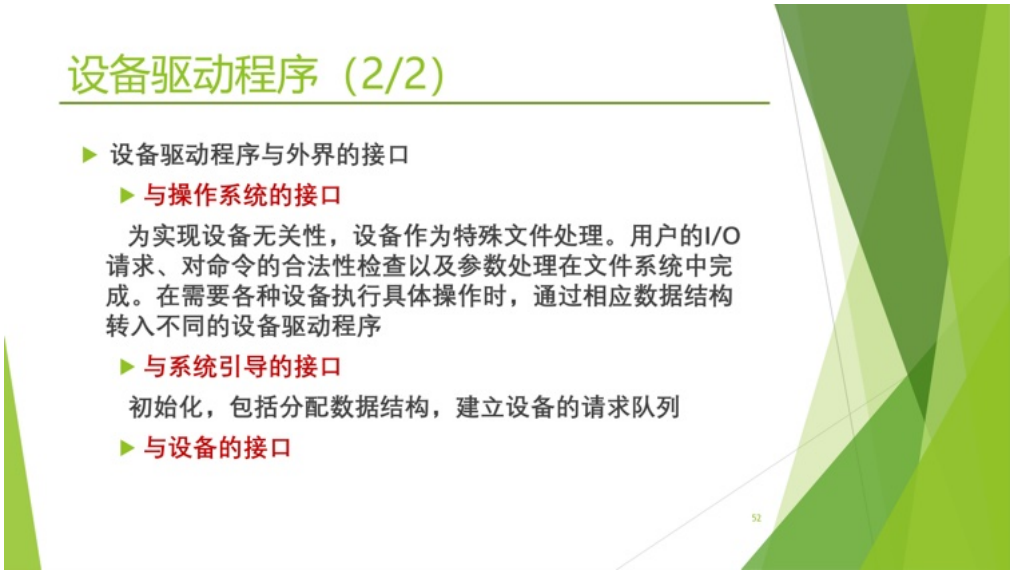
> **思考题：**设备分配为什么也可能造成死锁？

解答：设备是资源。如果进程占有一个设备又等待另一个设备，且设备不可抢占，就可能满足死锁四条件。例如进程 A 占有打印机等待磁带，进程 B 占有磁带等待打印机。设备分配必须考虑申请顺序、一次性分配、超时回收或 SPOOLing。

12.12 设备驱动程序与 I/O 进程

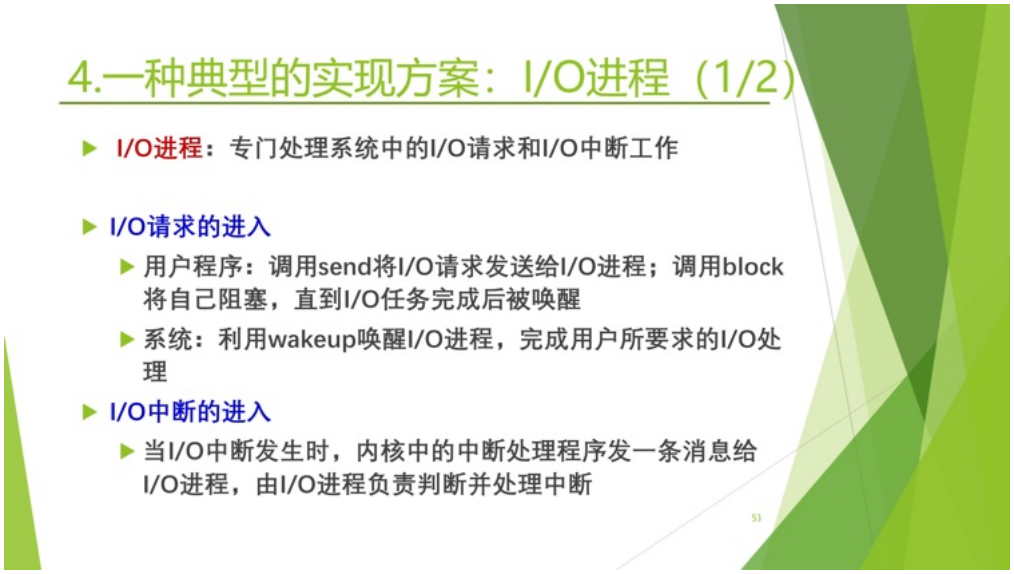
驱动程序职责：

- 初始化设备；
- 接收上层请求；



图示：设备驱动程序把设备相关代码封装起来，并通过统一接口连接上层。

- 参数检查和命令构造；
- 操作设备寄存器；
- 管理请求队列；
- 处理中断下半部或完成回调；
- 错误处理和重试；
- 向上层报告完成状态。



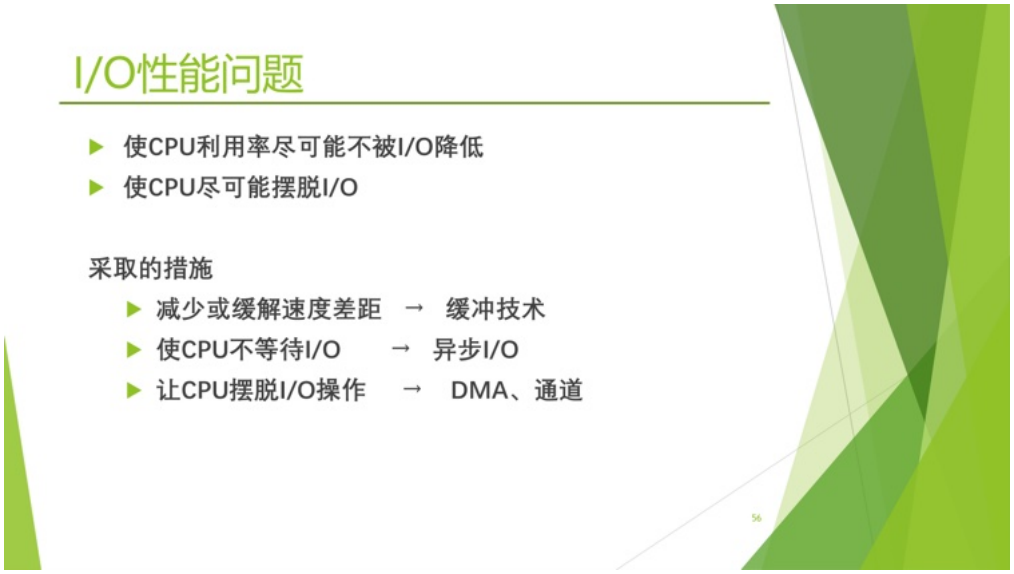
图示：I/O 进程专门处理系统中的 I/O 请求和中断工作。

I/O 进程模型：

- 系统进程，通常优先级较高；
- 接收用户或内核其他部分的 I/O 请求；
- 驱动设备运行；

- 中断到来后被唤醒处理后续工作；
- 有助于把复杂 I/O 控制流程进程化。

12.13 I/O 性能问题



图示：I/O 性能优化目标是让 CPU 尽可能不被 I/O 降低利用率。

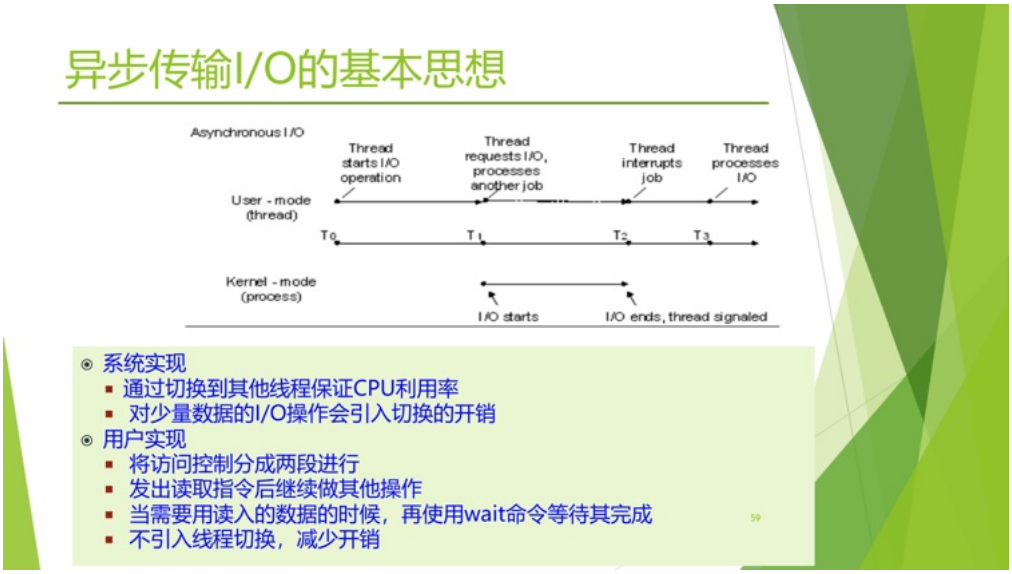
核心目标：

- 减少 CPU 等待 I/O；
- 减少中断和上下文切换；
- 增加批量传输；
- 提高设备并行度；
- 使用缓存和预读；
- 请求合并和重排序；
- 使用 DMA 和异步 I/O。

12.13.1 同步、异步与阻塞、非阻塞

- 同步 I/O：调用者等待操作完成语义。
- 异步 I/O：发起请求后继续执行，完成后通知。
- 阻塞：线程等待资源或结果。
- 非阻塞：暂时不能完成就立即返回。

这两组概念不同：非阻塞读可能仍是同步接口；异步 I/O 通常通过回调、事件、完成队列通知结果。



图示：异步 I/O 把发出请求和等待结果分离，提升 CPU 利用率。

12.13.2 异步 I/O

异步 I/O 基本思想：

- 用户发出读取/写入请求；
- 请求进入内核或运行时队列；
- 用户线程继续执行其他工作；
- 设备完成后通过信号、回调、事件、完成端口或轮询队列通知；
- 用户再收集结果。

适用场景：

- 网络服务器；
- 高吞吐存储；
- GUI 程序避免主线程卡顿；
- 数据库和日志系统。

12.14 本讲小结

- I/O 管理难在设备差异大、速度慢、错误复杂、并发多。
- 设备控制器通过寄存器和中断连接 CPU 与外设。
- 轮询简单但浪费 CPU；中断减少轮询；DMA 减少 CPU 数据搬运。
- I/O 软件分层把设备差异封装在驱动中，向上提供统一接口。
- 缓冲技术解决速度不匹配，是 I/O 系统最基本的优化。
- 设备分配也会产生死锁，必须考虑资源管理策略。
- 异步 I/O 的目标是把等待设备的时间转化为 CPU 可执行其他工作的时间。

—— 笔记完 ——

北京大学　操作系统　2026 春